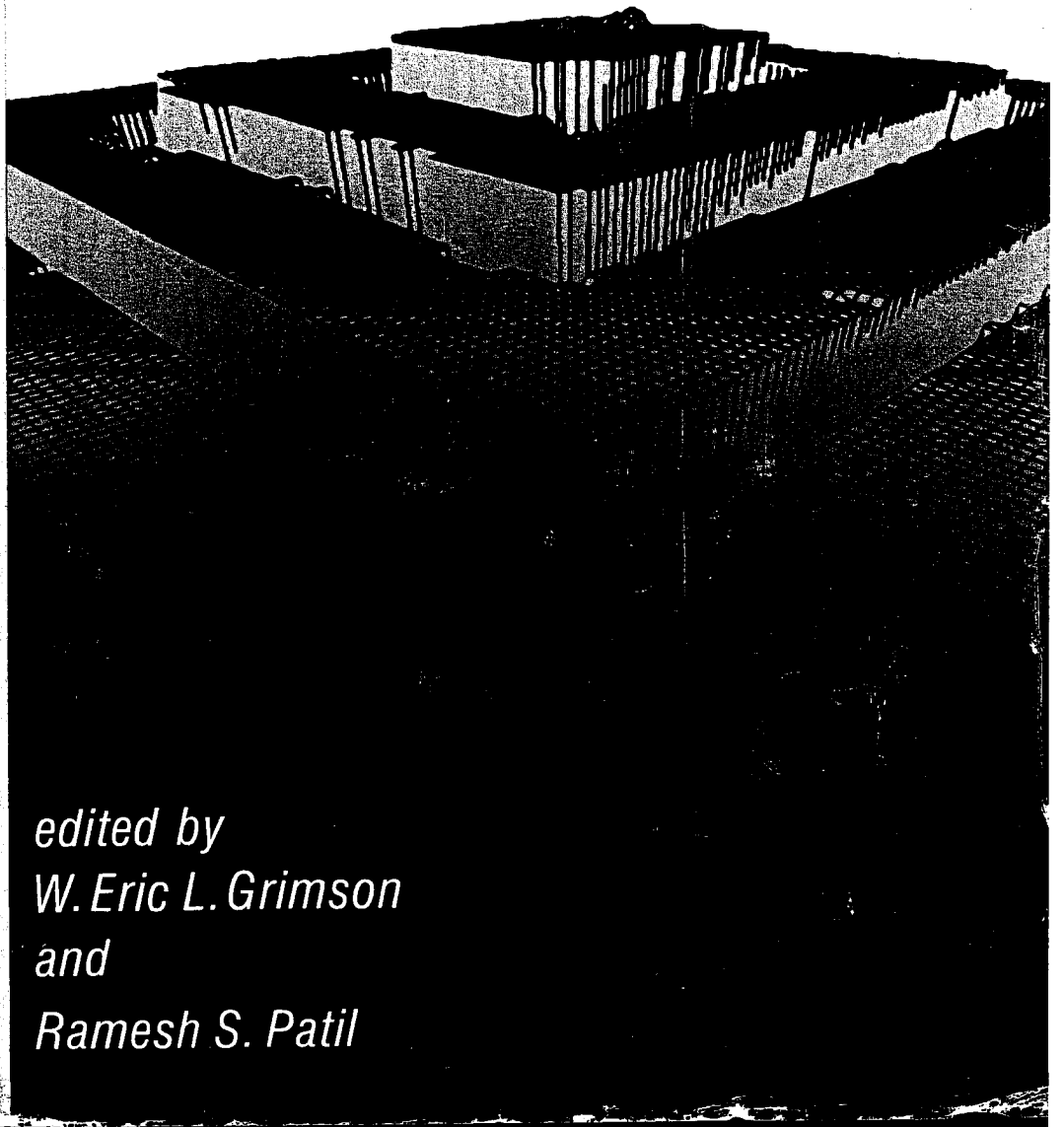
AI in the 1980s and Beyond

An MIT Survey



edited by
W. Eric L. Grimson
and
Ramesh S. Patil

AI in the 1980s and Beyond

The MIT Press Series in Artificial Intelligence
Edited by Patrick Henry Winston and Michael Brady

Artificial Intelligence: An MIT Perspective, Volume I: Expert Problem Solving, Natural Language Understanding, Intelligent Computer Coaches, Representation and Learning edited by Patrick Henry Winston and Richard Henry Brown, 1979

Artificial Intelligence: An MIT Perspective, Volume II: Understanding Vision, Manipulation, Computer Design, Symbol Manipulation edited by Patrick Henry Winston and Richard Henry Brown, 1979

NETL: A System for Representing and Using Real-World Knowledge by Scott Fahlman, 1979

The Interpretation of Visual Motion by Shimon Ullman, 1979

A Theory of Syntactic Recognition for Natural Language by Mitchell P. Marcus, 1980

Turtle Geometry: The Computer as a Medium for Exploring Mathematics by Harold Abelson and Andrea diSessa, 1981

From Images to Surfaces: A Computational Study of the Human Early Visual System by William Eric Leifur Grimson, 1981

Robot Manipulators: Mathematics, Programming and Control by Richard P. Paul, 1981

Computational Models of Discourse edited by Michael Brady and Robert C. Berwick, 1982

Robot Motion: Planning and Control edited by Michael Brady, John M. Hollerbach, Timothy Johnson, Tomás Lozano-Pérez, and Matthew T. Mason, 1982

In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension by Michael G. Dyer, 1983

Robotics Research: The First International Symposium edited by Michael Brady and Richard Paul, 1984

Robotics Research: The Second International Symposium edited by Hideo Hanafusa and Hirochika Inoue, 1985

Robot Hands and the Mechanics of Manipulation by Matthew T. Mason and J. Kenneth Salisbury, Jr., 1985

The Acquisition of Syntactic Knowledge by Robert C. Berwick, 1985

The Connection Machine by W. Daniel Hillis, 1985

Legged Robots that Balance by Marc H. Raibert, 1986

Robotics Research: The Third International Symposium edited by O. D. Faugeras and Georges Giralt, 1986

Machine Interpretation of Line Drawings by Kokichi Sugihara, 1986

ACTORS: A Model of Concurrent Computation in Distributed Systems by Gul A. Agha, 1986

Knowledge-Based Tutoring: The GUIDON Program by William Clancey, 1987

Visual Reconstruction by Andrew Blake and Andrew Zisserman, 1987

AI in the 1980s and Beyond: An MIT Survey edited by W. Eric L. Grimson and Ramesh S. Patil, 1987

AI in the 1980s and Beyond

An MIT Survey

Edited by W. Eric L. Grimson
and Ramesh S. Patil

The MIT Press
Cambridge, Massachusetts
London, England

PUBLISHER'S NOTE
This format is intended to reduce the cost of publishing certain works in book form and to shorten the gap between editorial preparation and final publication. Detailed editing and composition have been avoided by photographing the text of this book directly from the author's prepared copy.

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

AI in the 1980s and beyond.

(The MIT Press series in artificial intelligence)
Includes index.
1. Artificial intelligence. I. Grimson, William Eric Leifur. II. Patil, Ramesh S. III. Massachusetts Institute of Technology. IV. Series.
Q335.A42 1987 006.3 87-3241
ISBN 0-262-07106-1

Knowledge-Based Systems: The View in 1986

Randall Davis

Three things of interest stand out in reviewing the state of the art of knowledge-based systems in 1986. First, two technologies for building these systems — rules and frames — have become reasonably familiar and widely applied. This paper reviews them both briefly, exploring what they are and why they work, both as a way of characterizing what we currently know well and as lead-in to a discussion of some of their problems and limitations.

Second, we are seeing a strong surge of commercialization of the technology, leading to some interesting developments in the system-building tools that are being created, and an interesting evolution in the role of the technology within organizations.

Third, new ideas have begun to emerge about how to build these systems, ideas that are being referred to as model-based reasoning, or "reasoning from first principles." To illustrate them, I'll describe some of the work that's going on here in my group at MIT.

Knowledge-Based Systems

Any time a new technology appears, new jargon emerges to describe it. What are we to make of the term "knowledge-based system"? The term is really an answer to the question "What makes an expert an expert?" That is, why is someone who is good at what they do *that* good at it? Do they think differently than the rest of us? Do they think faster than the rest of

us? Is there something inherently different about an expert's thought and problem solving?

Calling something a knowledge-based system is in part a commitment to believing that that's not true. It is instead a commitment to believing that the primary source of expertise is knowledge, that problem solving skill arises primarily from knowledge about the task at hand. That knowledge includes both facts about the problem and a wide array of problem solving strategies that experts accumulate over time.[1] Programs built in this inspiration are knowledge-based in just this sense: their power arises not from their speed as computer programs, nor their ability to retain almost endless detail; it is instead based on their sizable store of knowledge about the task domain.

Rule-Based Systems

Figure 1 shows an extract taken from a pamphlet on tax planning, specifically selecting an organization structure to maximize the benefit from regulations concerning tax-loss pass-through. This extract, perhaps unwittingly, takes this very same view. Consider the first paragraph and note in particular the first three words: *Skilled practitioners know...*, reflecting the implicit view that people are good at a task because they know something about it.

> Skilled practitioners know how to qualify a business for pass-through treatment while avoiding potentially unpleasant consequences.
>
> There are two classes of partnerships: general and limited. General partnerships have the advantage of being much easier to draw up and of having no problems qualifying as partnerships for tax purposes. But they also have the significant disadvantage of establishing unlimited liability of the partners for the debts of the partnership.
>
> If the partners do not care about liability, or liability insurance is adequate, then a general partnership is easiest. If liability insurance is not adequate and the investors do not care about management participation, then a limited partnership is very likely the right idea.

Figure 1. Extract from a pamphlet on tax planning.

[1] Other factors affect performance as well, of course: things like practice, for example, allow the expert to perform smoothly and quickly. The claim here is that the primary, but not sole, foundation for problem solving skill is knowledge of the task.

Consider the second paragraph, which tells us facts about the world. In particular it tells us there are two classes of partnerships, general and limited, and then goes on to tell us some additional facts about each of those.

Finally, the last paragraph gives us some rules for thinking about and solving problems in this domain; in particular choosing one or another of these organizational structures to to take advantage of tax laws on handling business losses.

This text rather nicely reflects much of the mindset of knowledge-based systems: it notes the dependence of skill on knowledge and then illustrates two forms of that knowledge: facts about the domain and rules for solving problems in the domain.

We are in general accustomed to the idea that computers can be used to help do arithmetic: it's relatively easy to give a machine a formula for compound interest or use it to figure out budgets. We are somewhat less accustomed to the notion that we can take information like that in Figure 1, put *it* inside such a machine and have the machine use it in solving problems.

Part of the contribution of the current knowledge-based system technology is a way of doing this that is relatively easy to use. We can, for example, take the last sentence in Figure 1 and reformat it just a bit to make its content more explicit (Figure 2). As Figure 2 shows, the rule indicates that if three conditions are met, then there is some evidence that a limited partnership is the right choice.

IF

you are considering a partnership, and
liability insurance is not adequate, and
the investors do not care about management participation,

THEN

there is strong evidence (.8) that a limited partnership
is the right structure.

Figure 2. Simple if then-decision rule.

The important point here is that we have a way of capturing *symbolic inference rules*, rules of inference that can capture the thinking and problem

solving of experts. While mathematical models and calculations can be quite powerful, much of what we know about the world is not well captured by numbers, much of our reasoning is not well modeled by arithmetic. Rules of the form shown in Figure 2 can be of considerable utility for capturing and putting to use this other form of knowledge. They allow us to capture some of the ways people think about a problem and embody those ideas inside the machine.

Systems built in this manner are called *rule-based systems*; their standard architecture is illustrated in Figure 3. The two most fundamental components are the inference engine and the knowledge base. The knowledge base typically contains a few hundred to a few thousand rules of the very simple if-then decision form shown above. The inference engine is a mechanism that uses the knowledge in the knowledge base, applying it to the problem at hand.

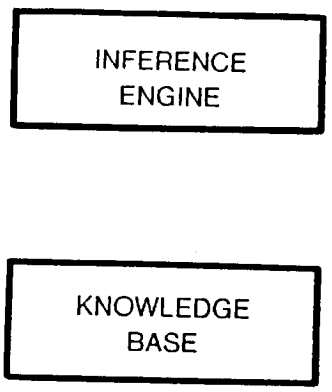


Figure 3. Architecture of a knowledge-based system.

Perhaps the most common way of using the knowledge is *backward chaining*.[2] Suppose we were trying to use rules of the sort shown in Figure 2 to determine what structure to select for an organization. The system will retrieve all the rules that are relevant to that conclusion, namely all the rules that tell us something about that particular topic. Rule 27 (Figure 2) is one such rule because it concludes about an organizational structure.

[2] For reasons of space the description here of this process is much abbreviated. For a more complete description see [Davis77].

We can make the conclusion shown in that rule if each of the three clauses in its "if" (or "premise") part are true. How do we determine the truth of each of these? We start all over again doing exactly the same thing, namely retrieving all the rules that are relevant to the first clause (about liability insurance) and trying each of those rules in turn (Figure 4b).

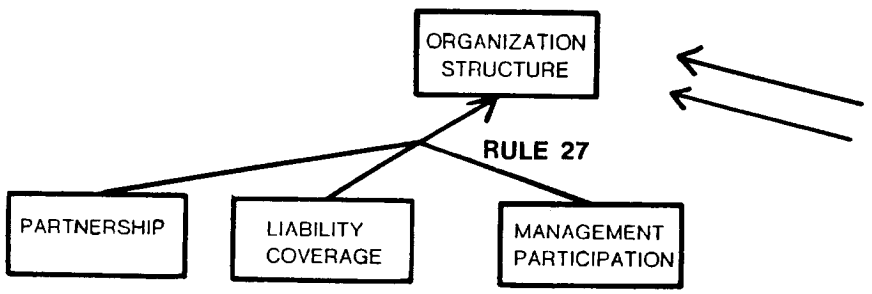


Figure 4. Backward chaining, step 1.

This is called backward chaining because we're starting with what we're after (organization structure) and retrieving rules relevant to that topic; this in turn brings up new topics we have to deal with. We then retrieve rules relevant to those new topics, working backwards toward ever more basic pieces of information. Eventually, of course, the systems encounters topics for which there are no rules. At that point it requests specific facts from the user in order to determine the truth or falsity of the premise clause under consideration.

Eventually the system will determine the truth of all three clauses of the rule; if all have been satisfied then this rule will fire and make one conclusion about organizational structure. Since rule 27 offers evidence that is strong but not definitive, the system will then go on to try all the other rules relevant to that topic, in effect collecting all the "opinions" it can about what organization to choose.[3]

From the user's point of view, interaction with the system looks like the trace shown in Figure 5. The system is working its way backward through the rules, encountering topics for which it has no rules, asking questions of the user via the dialogue shown here.

[3] See [Shortliffe75] for one description of how to deal with the uncertainty captured in phrases like *strong evidence* and *weak evidence*.

1) What is the expected startup income (loss) of the business in the first fiscal year?
** (182000)

2) What is the investor's marginal tax bracket?
** 50

3) Assuming losses can be passed through, will the losses be useful to the investor?
** Y

4) What is the business manager's expected marginal tax bracket?
** 45

5) Is the business liability coverage adequate?
** UNKNOWN

Figure 5. Sample trace of the system in operation.

There are two particularly interesting characteristics of these systems. First, they work. It was not obvious when efforts began in this area more than a decade ago that one could in fact capture non-trivial bodies of expertise by collecting somewhere between a few hundred and a few thousand very simple rules of the form shown. It's an empirical result, and a somewhat surprising one, that it simply turns out to work and in fact works for a fairly wide variety of applications.

The second interesting characteristic of these systems is shown in Figure 6. Here, in the middle of another consultation, instead of answering the question the user has asked "why," that is, "Why are you asking me this? Why does it make sense to request that piece of information?"

The system's answer is shown, saying in effect, "Here's what we're after (organization structure), we've already determined that the first two items are true, so if we also know this third item, then we'll be able to make the conclusion shown."

There are two interesting things going on here. First, we have in effect stopped the program in the middle of what it was doing, said "What are you doing?" and the answer that came back was comprehensible. It's comprehensible in part because, in the most general sense, what the program is doing is symbolic inference, not arithmetic; problem solving, not calculation.

7) Might the investor want to manage the enterprise at some point?
** WHY

We are trying to determine the right structure for the organization.

It has already been determined that
[1] you are considering a partnership, and
[2] liability insurance is not adequate

Hence if it is also true that
[3] the investors do not care about management participation,

then we can conclude that there is strong evidence that a limited partnership is the right structure.
[RULE27]

Figure 6. Answering a "why" question.

Imagine by contrast stopping a program that does linear programming or Bayesian decision theory or some other variety of fundamentally mathematical operation and saying "What are you doing?" What could it possibly say?

Mathematical methods, as powerful as they are, and as appropriate as they are for a number of problems, simply are not and never were intended to be models of how people think. One of the advantages of a knowledge-based system is that it is so structured, that it does allow us to capture the way people reason.

The second interesting point is *transparency*. The program is not a "black box." We can in effect look inside it and see how it works, see what it's doing, examine the line of reasoning being used.

That in turn is important because it leads to a notion of *accountability*. The system does not simply supply an answer, but can in addition tell us something about why that answer seemed appropriate. Hence in addition to getting a result, we can also get an accounting of how that result arose. As a result, we need not accept the answer on a take it or leave it basis; we can instead examine it in the context of its explanation and come to understand both the outcome and its rationale.

Figure 7 shows another form of explanation. At the end of a consultation the system will display its answer about how it thinks the organization ought to be structured. We might then discover that the answer we would

have expected to see in fact doesn't appear. It is possible at that point to ask a question of the sort shown in Figure 7. We are in effect asking the system why it didn't do something else.

** WHY DIDN'T YOU SUGGEST AN S CORPORATION?

There are three rules that indicate that the structure of the organization should be an S corporation (rules 98, 43, 17).

However, all of them require that the number of shareholders is no more than 35. You said (in answer to question 24) that the number of shareholders is 43.

Figure 7. Answering "why not".

It responds by telling us which rules could have come to that answer, then explaining why none of them were applicable. Several things are worth noting here. First, in addition to *using* its knowledge to solve the problem at hand, the system is also capable of *examining* that knowledge. The system isn't applying the rules here (as it did to generate its original answer), but is instead examining the knowledge base it to determine how it might have arrived at the alternative answer.

Second, this very simple form of introspection gives us a very interesting consequence: the system can not only tell us what it did, it can tell us what it *didn't do and why.* Consider the consequences of that in the context of due diligence.

Third, this is another illustration of the notion of accountability: the system is not only indicating what it did, but is explaining as well why other alternatives were not appropriate.

Finally, consider in the long term the impact of systems like this in decision support: they not only facilitate the decision making process, but can as well facilitate the communication of the reasoning behind the decisions.

Most of the practical application systems built to date have been constructed using rules, in part because this is the simplest and best understood technology, though it is by no means free of shortcomings. We consider limitations of the technology below.

Frame-Based Systems

The second fairly well-established approach is based on the idea of *frames*

as a way of expressing knowledge. Since they were first suggested in rather high level terms some years ago [Minsky75], frames have come to mean a number of different things and have been used in several different ways. Despite the diversity, four basic concepts appear common to the different conceptions:

- A frame contains information about a prototypical instance.

  In medical applications, for example, a frame is used to describe a prototypical or "classic" case of a disease.

- Reasoning with frames is a process of matching prototypes against specific individuals.

  For example, a specific patient is matched against a collection of disease prototypes to find the closest match.

- Frames are often organized into taxonomic hierarchies, providing an economy of knowledge representation and reasoning mechanism.

  A small segment of a diseases hierarchy is shown in Figure 8; we consider below how this can be used.

- The information within a single frame is typically expressed as a collection of *slots*, each of which has one or more *values* in it.

  This aspect of frames offers a convenient way of expressing many of the simple facts about a domain.

The knowledge base for a system like this typically consists of a large number of frames, each capturing a single prototypical description. In a medical system, for example, a frame would describe one disease, such as a classic case of viral hepatitis.[4] Part of the task of building the knowledge base is to determine what the prototypes are (e.g., what all the diseases are), and then how best to characterize each (e.g., what the presenting symptoms or other characteristic features are). This is typically a large and difficult undertaking.

The basic task is then to match the set of frames against the collection of facts about a specific patient, to determine which frame or frames best matches. In more general terms, we need to compare the prototypical cases against a specific instance to determine the closest match.

The matching process proceeds in two phases: generating hypotheses by reasoning from observed symptoms to possible diseases, then evaluating

[4] As in the case of rule-based systems, the description given here of frame-based reasoning and representation is necessarily abbreviated. For a description of a real frame-based system, see [Pople82]; for more on the uses of frames for building expert systems, see [Fikes85].

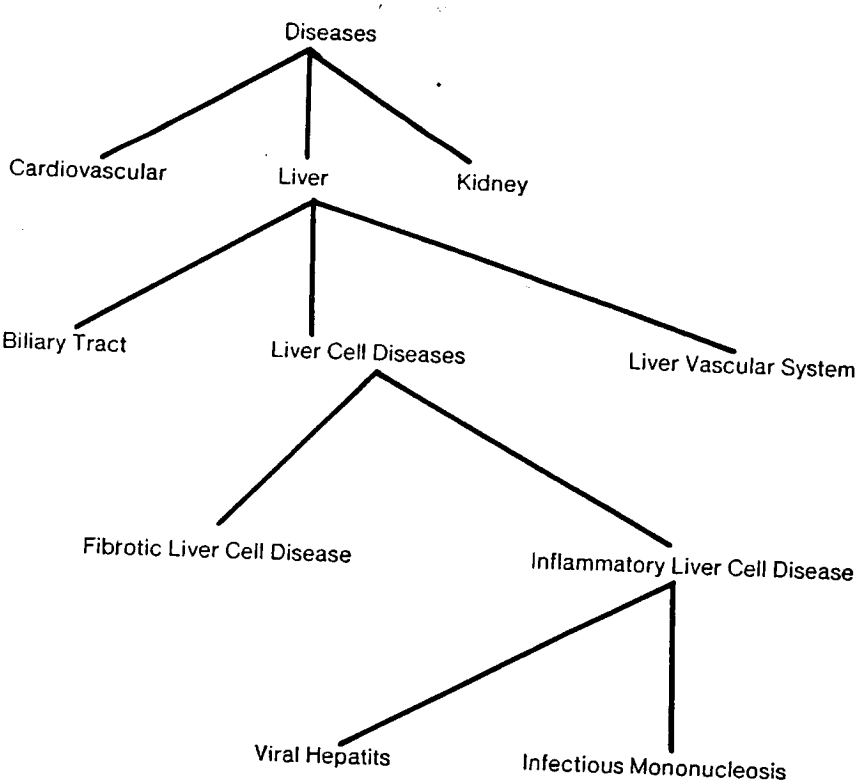


Figure 8. Segment of a disease hierarchy.

hypotheses by determining how well each accounts for the observed symptoms. In the hypothesis generation stage we have a collection of symptoms and need to ask what diseases they suggest. The basic question is, how well do the observed symptoms *support* any particular hypothesis?

One illustrative (but very simple) way to answer this is simply to count symptoms: if the patient has nine of the ten classic symptoms of viral hepatitis, we can be fairly confident that it is a plausible hypothesis. If, on the other hand, the patient only has one of the ten symptoms, we're much less likely to entertain that hypothesis.

The second stage goes the other way around. The question now is, given the diseases hypothesized, how well does each *account for* the observed symptoms? That is, suppose we believe for the moment that the patient actually has viral hepatitis. What consequences does this have for the set of observed symptoms? Does that account for all the symptoms, or

is there "undershoot", i.e., are there still some "left over"? If so, then we should start the process over again, this time matching the frames against only the "left over" symptoms, to see what else may be present.

There may also be "overshoot": are there classic symptoms of viral hepatitis that ought to be present and that don't show up in the current patient? If so, then the hypothesis overpredicts and we need to decide how to respond. One possibility is to rule out viral hepatitis: if the predicted symptom is a necessary finding in anyone with the disease, then its absence rules out the possibility of the disease.

Organizing frames into a taxonomy can add two things to this process. First, the concept of *inheritance* offers an economy of representational mechanism. It is based on the principle that many things true of a general category are true of the more specific subcategories as well. In Figure 8 for instance, many of the things true of liver cell disease are true for all subcategories of that disease as well. The basic insight is to store those things only once, as "high up" in the taxonomy as possible. We need not repeat in our description of viral hepatitis all those things also true of its supercategory, inflammatory liver cell disease. Viral hepatitis will "inherit" those from its supercategory.

This is relatively easy to accomplish in simple cases, but the general case is fairly difficult: there are numerous subtleties to inheritance, such as handling exceptions, that are not yet well understood.[5]

A taxonomy also offers the opportunity to reason by successive refinement: the system can begin by using frames near the top of the hierarchy, in effect classifying into a broad category, then refine this by using frames found at the next level beneath that category. This permits a focusing of effort that can be very important in a large knowledge base.

This is a much simplified version of how such programs work, but it gives a general sense of their operation. Real systems typically have considerably more sophisticated ways of generating and evaluating hypotheses, dealing with interactions between hypotheses, etc.

Roles, Successes and Limitations

Names can at times be problematic and that has been the case when these systems are referred to as *expert systems*. The term conjures up a wide variety of expectations about performance and role, for example. It is im-

[5] For more detail on this, see [Etherington83].

portant to note therefore that they need not function as experts in order to be very useful. There is in fact a whole spectrum of roles that are relevant: these systems can and have been built to function as both assistants and colleagues.

For any system in the foreseeable future, we will have a combination of human and machine. The real issue is, where on the line of collaboration (Figure 9) are we going to draw the separation of responsibility? If the line appears nearer the "machine" end, with the human responsible for most of the problem-solving, the system will be functioning as an assistant. If the line appears more toward the middle, it becomes a collegial relationship with both contributing equally. If the line can be pushed nearer to the human end, with the machine taking on most of the task, then we have something that may truly deserve the term expert system.

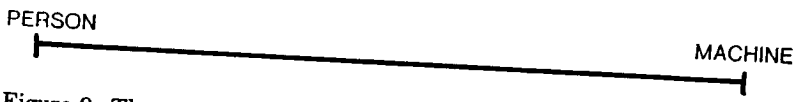


Figure 9. The spectrum of roles for a knowledge-based system.

The important point is that there are extensive pragmatic benefits at every point along that spectrum. The system need not be as good as an expert to be useful; even an intelligent assistant can be of great utility. Such a system can serve to offload routine cases, freeing up a human expert's time to focus on the more difficult and challenging problems. It in effect becomes an "intelligence amplifier", allowing us to use the scarce resource of the expert's time in a much more effective manner.

It's also important to recognize that there are a number of well known limitations in this technology. An extensive list appears in Figure 10; a few will serve to illustrate. There is, for example, the problem of scope: we have to define these problems narrowly enough to make the knowledge base construction task feasible, yet ensure that we don't define away the difficult part of the problem. If the task is *too* narrowly defined, all of the problem solving is done in simply knowing enough to come to this particular source of expertise.

There are also substantial difficulties surrounding the problem of inexact reasoning, problems recognized in mathematics and psychology some

Scope
How much of the problem is solved in coming to the expert?

Getting Religious
Some information is not well captured by rules of frames.

Simple Representation
The standard attribute/object/value representation is very weak.

Multiple Diseases
Most systems assume no interaction between them.

Inexact Knowledge
A difficult problem, both definition and execution.

Common Sense
Pervasive in human reasoning, yet elusive.

Accumulating Knowledge By Cases
Is there a more fundamental understanding than individual rules for individual situations.

Character Of The Knowledge
Reasoning qualitatively and reasoning causally are both important in human reasoning, but difficult to capture.

Learning
Wouldn't it be better to just have the system discover what it needed to know.

Figure 10. Limitations of the current technology.

time before knowledge-based systems arrived. One fundamental problem is to specify exactly what is meant when a rule says "probably", "maybe", or "very likely." Does this refer to probability or to strength of belief? That is, does the rule indicate how likely it is that something is true, or does it refer to how strongly the rule author believed it? The two are quite different, as any gambler knows.

A second difficulty lies in defining the "arithmetic" to use in combining them. What does "maybe + probably" add up to? Very likely? Very likely. The problem has been attacked by mathematicians and philosophers; one indication of its difficulty is the large number of solutions that have been suggested.

As one final example, there is the difficult problem of learning. Systems today are typically built by debriefing experts, in effect by asking them to tell us what they know about the task at hand. This is difficult, time consuming, and not incidentally, assumes that there exists a willing expert.

One clearly desirable alternative is automating induction: we would like to be able to take a large collection of case studies — example problems with answers — give them to the program, and have *it* figure out what it ought to know. It's a fine aspiration, but a difficult problem, with much active and exciting research going on right now. But the message here is that this is still a long term dream: current practice on any non-trivial problem is still a process of finding a willing expert and spending a long time talking.

Commercialization of the Technology

A second major issue worth note in reviewing the current state of the art in knowledge-based systems is the fast-growing commercialization of the technology. One of the most interesting trends in this area is an evolution in the perception of the technology from something used strictly in support of existing manufacturing businesses, to one that is providing the basis for new products and new businesses.

The earliest applications of knowledge-based systems have been as in-house production aids. Systems like XCON, ACE (built by AT&T for cable maintenance), and others are designed to help the corporation carry out its existing function, perhaps faster and/or cheaper, but still in pursuit of its traditional agenda.

More recently we are beginning to see the development of systems designed not for internal consumption, but for sale as finished products, at times allowing an organization to move into a new line of business. An important characteristic of these systems is that AI is the supporting technology, not the product. Unlike the general purpose tools (expert system "shells" and other toolkits) currently appearing in the market in large numbers, they are designed to meet specific applications needs that AI can support, rather than simply to offer the technology and leave the application to the user.

One example, interesting because it characterizes a growing class of such systems, is a program called **PLANPOWER**^{TM}, built by Applied Expert Systems (Cambridge, MA) to do individual financial planning. More generally, the task is matching consumer product needs against an exploding marketplace of choices.

The need is particularly acute in financial services. New products appear with distressing frequency (zero-coupon bonds, index funds, sector funds, etc.). As a result of deregulation new players are entering the game, with banks selling life insurance, brokers selling CD's and insurance salesman selling tax shelters. The profusion of new products is overloading the traditional financial advisor, while the new players are hard-pressed to find enough trained personnel to provide the desired level of service.

PLANPOWER embodies the financial planning skill of expert human planners, discovered through extensive debriefing, aimed at finding out how to match needs against available products, and how to plot a course from where the client is at the moment in their financial life to where they would like to be in the future. The system functions in the traditional decision support system role, not replacing, but leveraging the time of financial planning professionals. Time saved in backoffice plan preparation translates directly into more time spent with clients and more time available to spend with new clients.

The application is also interesting because it involves combining both traditional arithmetic calculations (e.g., cashflow, tax projections, etc.), with the judgemental knowledge characteristic of knowledge-based systems (e.g., knowing what kinds of financial planning strategies to try).

Most important, however, are several interesting characteristics in the design and conception of the system. It is designed for use by service professionals, not by engineers, manufacturing or technical staff. It is designed for sale to and use by outside organizations, not for internal use and con-

sumption by the organization that created it. And it permits organizations to enter new markets by offering a way to distribute expertise to a large field force. For the traditional players in the market, it offers a way to deal with the information explosion that overloads their existing service providers. For the new players, it offers a means for building a large and highly skilled service network quickly and relatively inexpensively.

There are other, somewhat more speculative consequences that we may see in the long run. It may prove possible to use such systems to distribute corporate policy. Suppose an organization decides to make a significant change in its approach to a problem or a market. It can send out memos, the usual 10-100 page documents, hoping that someone reads them. In time, it may instead be able to send out new floppy disks, reloading and updating the knowledge base in the field, in one quick and effective step changing the way the entire field staff approaches the problem.

This is also one example of information technology used as a strategic weapon. Given, for example, the typically extensive databases compiled by financial planners about their clients, new product marketing strategies become possible, including automatic, very narrow targeting of new products to a carefully selected subset of the customer base.

What are we likely to see in the near future? More systems like **PLANPOWER** will appear, that is, more systems designed for non-technical, service-oriented end-users, systems that permit companies to attack new markets and provide new services.

The system building tools are going to get better. As useful as they are currently, it still takes a non-trivial amount of skill to use them well, in part because the art of system building is still new enough that we don't know how to make it a routine operation accessible to anyone.

We're going to see many "mundane" applications. Despite the image of AI as high technology, it may be that the most commercially significant applications are hardly the stuff of science fiction. Campbell's Soup, for example, was faced recently with the imminent retirement of someone with 44 years of experience in running one of its soup production lines. Rather than lose that body of skill, they worked with the expert to capture what he knew about that specific task and embodied it in a rule-based system. Running a soup cooker well is far from dazzling high technology, but it does save significant time and money. I think we're going to see many more of these apparently mundane but in fact quite important applications.

Finally, and, I hope, quite soon, we will see a much more subdued

acceptance of the technology. AI and knowledge-based systems are exciting, fundamentally new tools, but in the long term they need to be added to the existing kit of techniques for applying information processing. Much as with personal computers, spreadsheets, and other technologies, they have both their important roles and their limits. The sooner they are accepted as such the better.

Reasoning "From First Principles"

The final major issue I'd like address is the technology that we believe will provide the foundation for the next generation of knowledge-based systems. I'll do this by examining some of the research currently going on in my group at MIT.

The fundamental issue is solving problems in a style called reasoning "from first principles." Perhaps the easiest way to understand this is to contrast it with traditional knowledge-based systems, paying particular attention to the character of the knowledge captured in each.

Traditional systems are built by identifying and accumulating rules of the sort discussed earlier, rules that can be characterized as *empirical associations*. They are connections, typically between symptoms and diseases, which are "true" in the sense that they have been observed to hold empirically, but typically can't be explained in any more detail.

As one example, consider the following rule that we might find in a medical diagnosis system.

If the age of the patient is between 17 and 22
the presenting complaint is fatigue,
the patient has a low-grade fever,
then the disease is likely (.8) to be mononucleosis.

In some sense this rule is "true"; doctors have noticed the association often enough to know that it is is worth taking note of. But if we ask *why* it is true, we often find that the answer is something like, "The ultimate etiology is yet to be determined," (medical jargon for "Damned if we know").

The important points here are the associational character of the knowledge, its lack of more fundamental explanation, and it typical origin: the accumulated experience of practitioners. Such rules can be an effective way to capture expertise in domains where there does not yet exist a more fundamental understanding of cause and effect.

Suppose now that we tried to used this rule-based approach to do computer diagnosis in the literal sense: i.e., hardware maintenance, determining what's wrong with my computer when it breaks.

A number of difficulties would appear. First, the set of rules would be strongly machine-dependent. The particular set of foibles and symptoms common to one model of computer are often quite different from those of another, meaning the work of accumulating such rules must begin again for each machine. That in turn presents a significant problem, because the design cycle for new machines is growing short enough that there may not be time to accumulate all the needed rules. Field service staff will as a result constantly be catching up, struggling to stay ahead of the introduction of new machines as their recently acquired rules soon become obsolete.

The same problem crops up in a less dramatic way in the design upgrades made to machines already in use. The changes to the design may be quite small, but this can lead to significant changes in the surface symptoms, the way in which problems manifest themselves. Once again, the problem is rapid obsolescence of the knowledge.

The traditional approach has been reasonably successful in medicine because the model line is quite limited (only two basic models) and it has been relatively stable over the years (the design cycle is rather longer). As a result we have had the time to accumulate the relevant knowledge base, one that remains reasonably stable.

Finally, there is the problem of novel bugs. Traditional rule-based systems accumulate experience, embodying a summary of all the cases the expert has seen. If a problem presents unfamiliar surface manifestations, by definition the rule-based approach will be unable to deal with it, even if the underlying cause is within the scope of the system's competence (i.e., the system can identify it given other, more familiar symptoms).

Troubleshooting

For all those reasons the traditional rule-based technology was found to be insufficient to attack the problem of troubleshooting electronic hardware. Instead we sought to reason from a description of structure and behavior, in the belief that there were fundamental principles that could be captured. Such a system would have a degree of machine independence, would be quickly deployable, given the relevant schematics, and could reason from any set of symptoms, familiar or novel.

The specific problem of troubleshooting is part of a larger research strategy (Figure 11). Sitting at the center are languages for describing structure and behavior. Around the outside is a whole range of specific problems we are working on, including troubleshooting, test generation, test programming, and design debugging.

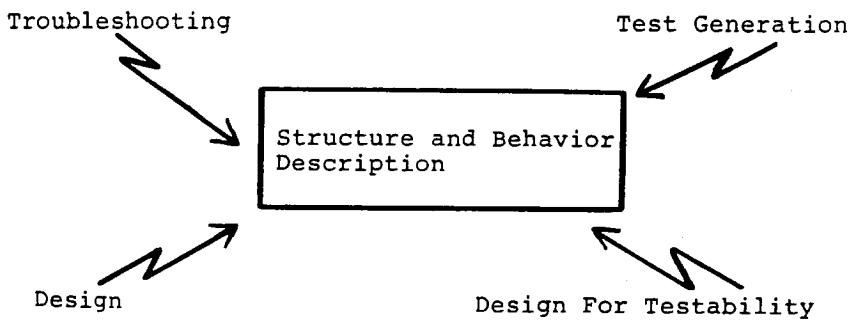


Figure 11. Overall research strategy.

The best way to understand our approach is to consider one specific example. The description that follows is a much abbreviated version of an example that appears in complete detail in a special issue of *Artificial Intelligence* [Davis84].

Consider the electronic circuit pictured in Figure 12, made up of three multipliers and two adders. It doesn't do anything particularly useful except provide a simple example to explain some of the interesting and difficult aspects of this problem.

Given the inputs shown, straightforward simulation will indicate that both outputs should be 12. Now suppose we also had a physical model of the device, gave it the same inputs, and measured what came out. Suppose that at output G we find a 12 as expected, but get a 10 at output F. Let's focus on this discrepancy between what we expected (from simulating the correct behavior) and what actually appeared, attempting to discover which component(s) might be broken in such a way as to account for the symptoms.

One way of attacking this problem by reasoning from structure and behavior starts by asking an obvious question: Why is it that we expected a 12 at F? There are three reasons: (a) we expected the box labeled ADD-1 to act like an adder, (b) we expected its X input to be 6, and (c) we

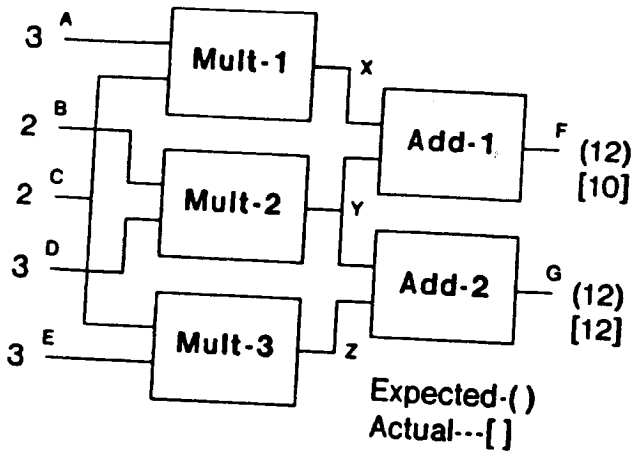


Figure 12. A simple troubleshooting example.

expected its Y input to be 6. If all those three things had been true, the value at F would have been 12. So one of them must be false. (To be more precise, *at least* one of them must be false, but we'll deal with just one for the moment.)

Consider each of those assumptions in turn. Suppose only (a) is false: in that case both X and Y are 6 (by assumption), but ADD-1 is not functioning as an adder. We don't know what it *is* doing, all we know are the symptoms (both inputs are 6, output is 10). Hence one possibility that is globally consistent with all the symptoms is that ADD-1 is broken in the fashion noted.

Now suppose only (b) is false. In that case, by assumption, ADD-1 is functioning properly and its Y input is 6, and we measured the value at F to be 10. Under those circumstances we can infer that the X input of ADD-1 must have been 4. That's interesting, because now we have another discrepancy, this time between the 6 predicted by simulation as the result of MULT-1, and the 4 inferred from the output observed at F. We can now ask the same question all over again ("why did we expect a 6 at X?"), and

discover that the problem may be a malfunction in MULT-1 (inputs 3 and 2, output 4).

Finally, suppose only (c) is false. In this case ADD-1 is working and the value at X is 6 (by assumption), the value at F has been measured to be 10, so we infer that the value at Y must be 4. Hence the third possibility is that MULT-2 is broken (inputs 3 and 2, output 4).

But while that's locally consistent with the value at F, it contradicts the value observed at G. That is, there is no way that MULT-2 alone can malfunction to produce the values observed. We conclude as a result that, given the symptoms shown, the only plausible single-fault candidates are either ADD-1 or MULT-1.

Though this reasoning is very simple, it demonstrates an important principle: knowledge of structure and behavior can form the basis for a powerful kind of diagnostic reasoning. Knowledge of structure is evident in the way in which the reasoning follows the connectivity of the components, moving from F to ADD-1, back through its inputs, etc. Knowledge of behavior was used both for the initial simulation, and later to infer inputs from outputs (e.g., we need to know how an adder behaves to infer the value at X given the values at Y and F).

As it turns out, that principle has some very interesting limitations. Consider Figure 13, with exactly the same circuit, but a new set of symptoms. What can be malfunctioning in this case? The simplest and most direct answer is that the second multiplier is malfunctioning by producing a 0 as its output. The process used just a moment ago will quickly derive this answer.

Interestingly, that's not the only answer. It might also be the case that the third multiplier *alone* might be broken and still account for the symptoms. How could that be?

The answer lies in some interesting assumptions we made about the device. Imagine that MULT-3 is packaged as a single chip all by itself. When chips are plugged into their sockets it sometimes happens that one of the pins doesn't make it into the socket. If that happens to the pin that supplies the power to the chip, then in addition to sending out a zero along its output, the chip can also exhibit the slightly odd behavior of "sending a zero out" along what we thought was going to be an input. More precisely, it can "drag down" its input. (An identical argument can be made for MULT-1 alone as the malfunction.)

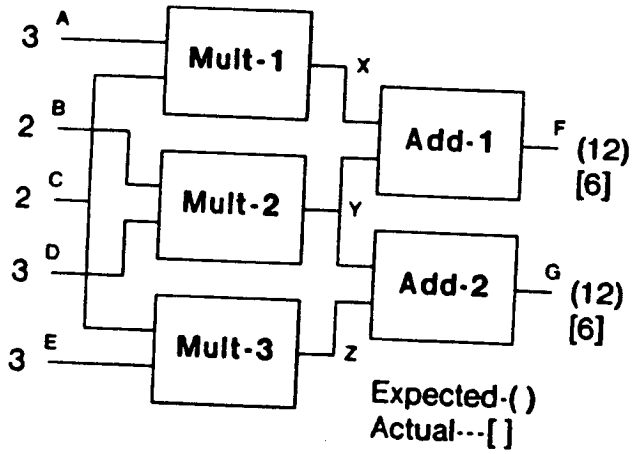


Figure 13. A variant on the first example.

The diagnostic process doesn't seem quite so intuitive anymore. To see one more example of the complexities that can arise, consider Figure 12 once again. Recall that we began the diagnosis by starting at the discrepancy at F and traced back through the connections shown, leading us first to ADD-1, then MULT-1 and MULT-2.

But why do we follow *those* connections? We might for example have said, "perhaps there's another wire, not shown in the diagram, that connects F to the bottom input of ADD-2", and then traced from F down to that input. We don't do that, of course, because the diagrams we are given are always complete and correct.

Or are they? Of course not. The point in making the claim about completeness and correctness so boldly is to make clear how tenuous it really is. There are in fact many ways in which the actual structure of the circuit can differ from the diagram we are given. Errors can occur in assembly: the device may simply be wired up incorrectly. Somewhat more common are problems known as bridging faults. Occasionally, when the chips are soldered into place, instead of small dots of solder at each pin, a

larger puddle forms that is big enough to touch two pins at once, forming a solder "bridge" between them. In effect an extra wire has been inserted between those two pins, a wire that cannot show up on the diagram because it is not supposed to be there.

In the face of this, our original technique of tracing signals through the network seems to be facing serious difficulties. The great virtue of that technique is that it reasons from a knowledge of structure and behavior. But its fatal flaw is that it reasons from a knowledge of structure and behavior. More precisely, it reasons from knowledge of the *intended* structure and behavior, and sometimes the fault lies precisely in the discrepancy between what was intended and what was constructed.

The real question then is not so much how do we trace through the networks we are given, as how do we know what network to trace. That, in turn, seems to open up a seemingly limitless set of difficulties. If we're trying to reason from knowledge of structure and behavior and we can't rely on the description given, what can we do?

We appear to be caught on the horns of a fundamental dilemma: to get anywhere on the problem we need to make *some* assumptions about how the device is structured. Yet any assumption we make could be wrong.

The fundamental dilemma is the tradeoff between complexity and completeness. For the diagnostic routines to be complete, they must consider every possibility, i.e., make no assumptions. But if they do that, they drown in complexity, since they are reduced to exhaustive testing of behavior, which is computationally intractable. To handle the complexity we must make some assumptions, yet that eliminates completeness, since it might cause us to overlook the thing that happens to be wrong.

The problem is broader than electronic troubleshooting alone. Almost any form of real-world problem solving faces this basic issue. To get started on a problem we must make simplifying assumptions; yet to be good at that task we must not be blinded by our assumptions.

The three key ideas that we use to deal with this problem are all quite simple. First, Occam's Razor: start with simple hypotheses and only generate more complex variants as the problem demands it.

Second, make simplifying assumptions, but keep careful track of them and their consequences. When a contradiction arises, the important task is to determine which of the simplifying assumptions was responsible, eliminate that assumption, and try solving the new (slightly more complex) version of the problem.

The final idea is to use multiple representations. In Figure 12 we had a representation of the circuit organized *functionally*, i.e., it shows the various boxes organized according to their purpose in the device. A second way to represent that circuit is shown in Figure 14, where we see it viewed *physically*, as the chips might actually appear on the circuit board. In the case of a bridge fault, this physical representation of the circuit is particularly useful because it helps to suggest where unexpected "wires" might turn up. More generally, we believe, there is for each general category of fault a representation which is particularly useful in just this same sense. In part then, the task of building a troubleshooter for any particular domain involves looking for a set of appropriate representations.

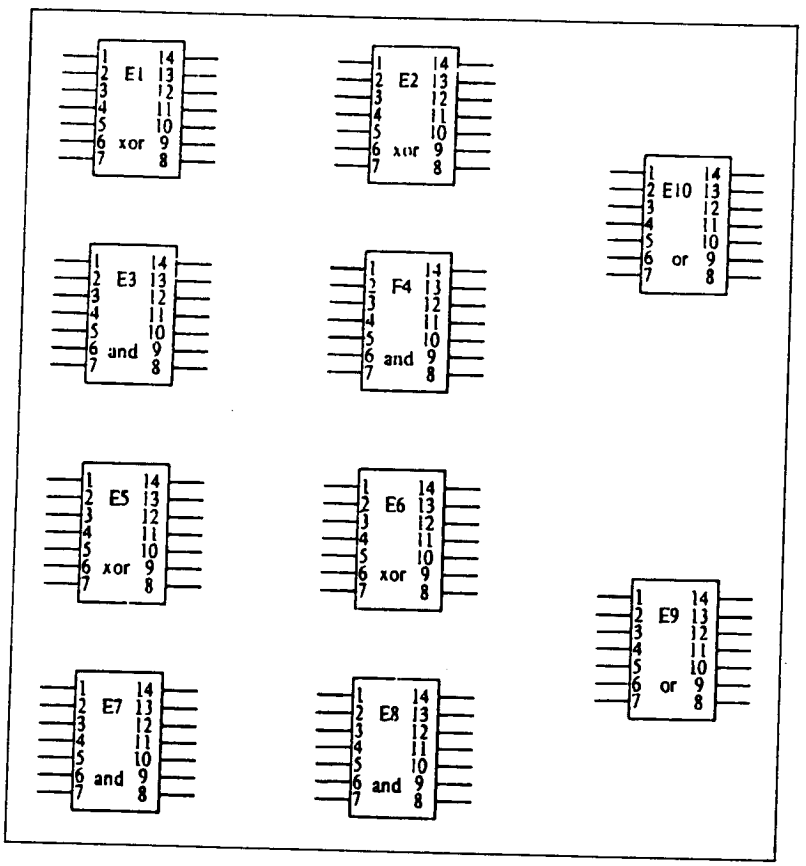


Figure 14. The physical view of the circuit.

These three ideas work together to support a fairly powerful form of diagnostic reasoning. The program can make simplifying assumptions in order to get started, back up when contradictions occur, and then try a slightly more elaborate version of the problem. The use of multiple representations helps by guiding the choice of the more elaborate version of the problem to try next.[6]

Design and Test Programming

The bulk of our experience to date has been on various forms of diagnosis, simulation and test generation [Davis84, Shirley83, Hamscher84, Haig86]. Two other, more recent foci of this line of research are design [Williams85] and test programming [Shirley86]. The fundamental task of design is, given a description of a desired behavior, specify a set of components and their interconnections that will produce that behavior. Test programming involves the creation of a set of tests, used at the end of the manufacturing line, that exhaustively verify the presence of all the intended behaviors of the device.

Current attempts at automated design share two important characteristics: they are largely *library-based* and done by *decomposition*. The basic paradigm of many automated design programs is to look in a library of standard designs to determine what subcomponents are needed, then look up designs for each of those, continuing until primitive components are reached. A design plan for a radio, for example, might decompose the task into designing a power supply, tuner, and audio section, then putting those three together. Given that basic plan, we can now look up a plan for designing each sub-component (i.e., power supply, tuner, audio section).

This can be quite useful and has provided the basis for some programs with interesting behavior. At their best these programs are capable of proposing designs that are novel in the sense that they involve previously unexamined combinations of library plans.

But there are two problems with this. First, there's something missing. There's something unsatisfying about saying that such a program "knows how to design a radio," when in fact all it did was look in a library of pre-existing plans. Second, this approach works with a fixed set of primitives. What happens if we need a tuner that has more sensitivity than is provided by any design plan in the library? All the approach can do is reject the plans

[6] Details of this process are found in the *AI Journal* article referenced earlier.

one after another, until it finally runs out. Since it does not in any sense "understand" any of the designs, it can do nothing to make fundamental changes in any of them.

We would instead like to be able to design something "from first principles", i.e., by understanding how all of its components work and how they contribute to the final behavior. This work, still in its early stages, is founded on several basic ideas [Williams85]:

- The ability of a designer to analyze a complex circuit appears to be based on a core set of very general principles. It may be the case that design, too, is founded on a core set of very general principles, many of which overlap principles used during analysis. Hence a "principled theory of design" involves the identification and use of such principles.
- Second, a theory of design must satisfy both robustness and competence. Robustness means that it is possible to design a wide class of systems based on the theory, while competence means that the design process should be efficient. A core set of design principles derived from analysis will provide robustness: such principles should make it possible to design any of the very large class of systems that is analyzable using the same principles.
- But design using those principles alone is very inefficient. Competence will arise from the use of four ideas: design experience, qualitative abstraction, approximation, and design debugging. Design experience refers to the accumulated experience of a community, captured as a set of design techniques commonly used by skilled designers. These represent shortcuts, well-recognized compositions of basic design principles.
- Qualitative abstraction and approximation allow the designer to focus on exactly those details that are relevant during a particular stage of the design process. These abstractions support a "least commitment" paradigm, enabling the system to delay all design decisions until the latest possible time, reducing the number of commitments that get made early and then later revised.
- Finally, the notion of design and debug suggests that it is at times considerably more enlightening to get the design "mostly correct" and then try it out, than to attempt to anticipate every detail. The system will recognize how and why a previous commitment turned out to be incorrect, then redesign based on what has been noted.

The task in test programming is not diagnosis of devices that have been working and suddenly begin to exhibit symptoms of misbehavior, but rather the complete verification of performance for a newly constructed device. The problem is particularly important in VLSI design because chip production has in recent years become increasingly test-limited. Where it was originally limited by problems in fabrication, and later by problems in complexity in design, it is more recently the case that we can design and fabricate chips complex enough that it is extraordinarily difficult to determine whether they actually work. Devices are becoming sufficiently complicated that none of the existing techniques for automated design of test programs is good enough. On complex chips existing test generation programs may never stop, running for weeks without producing an answer.

Interestingly, when this happens, chip manufacturers turn to human experts capable of solving the problem. Their skill in turn appears to rely on several interesting problem solving methods [Shirley86]. First, the experts remember many clichéd solutions to test programming problems. The difficulty in getting a program to this lies in formalizing the notion of a cliché for this domain. For test programming the clichés appear to contain fragments of test program code and constraints describing how program fragments fit together.

Second, experts can simulate a circuit at various levels of abstraction and recognize patterns of activity in the circuit that are useful for solving testing problems. An effective planning strategy can be based on symbolic simulation, coupled with recognition of the simulated events that accomplish the goal.

Third, and perhaps most important, this planning is guided by the knowledge that the device was purposefully designed to perform a set of specific tasks. The key here is that a device's *designed* behavior is far more limited than its *potential* behavior. This limitation translates into a reduction of the search necessary to achieve goals.

In addition to troubleshooting, design, and test program generation, we are also exploring tasks like design for testability. This is in some ways the dual problem to generating test programs: we can either design freely and then work hard to find a set of tests, or we can design with testability in mind and make the test generation task simpler. A wide range of applicable design techniques have been assembled over the years, but their use is for the most part a rather subtle art. We are attempting to codify some of that art and use it to build a system that will eventually be capable of

examining a proposed design and suggesting modifications only where the circuit is not currently testable.

Summary

In reviewing current technology we saw that rules and frames are two of the most common and best understood approaches to building knowledge-based systems. Rules capture informal, often judgemental heuristics, and are typically chained together, to form a line of reasoning leading to the answer. Backward chaining (from goals back toward more primitive data) is widely used because it mimics human problem solving; rules can also be run forward, making all logical conclusions from a basic set of input data. Frames capture prototypical situations and reason primarily by matching those prototypes against specific instances.

In examining today's commercial environment, we see a change in the kinds of systems being built. The original applications were all tools designed to solve in-house problems in production, quality, etc., all assisting in doing the existing business faster, better, or cheaper. More recently we are seeing the technology used to produce new products enabling strategic moves by organizations interested in new markets, and seeing products that are much more responsive to existing market needs, more market-driven than technology-driven.

Finally, we reviewed briefly some of the research currently underway at MIT, exploring the theme of reasoning from first principles, a theme we believe will be at the core of the next generation of knowledge-based systems. Its significance lies in part in its ability to provide an additional level of understanding, and as a result, an additional level of power in a whole range of tasks, including maintenance, test generation, circuit design and so forth. Our current work is aimed at digital electronic circuits because of the simplicity of the basic components, but we believe that the underlying ideas developed are applicable across a broad range of domains.

References

Davis, R., Buchanan, B., Shortliffe, E., 1977, " Production rules as a representation in a knowledge-based consultation system," *Artificial Intelligence*, 18, pp. 15–45.

Davis, R., 1984, " Diagnostic reasoning from structure and behavior," *Artificial Intelligence*, pp. 347–410.

Etherington, W., Reiter, W., "On inheritance hierarchies with exceptions," *Proc AAAI-88*, Washington, DC., pp. 104–108.

Fikes, R., Kehler, T., 1985, "The role of frame-based representation in reasoning," *Comm ACM*, pp. 904–920.

Haig, H., 1986, A language for describing digital circuits, MS Thesis, MIT, Computer Science.

Hamscher, W., Davis, R., 1984, " Diagnosing circuits with state, an inherently underconstrained problem," *Proc AAAI-84*, Austin, Texas, pp. 142–147.

Minsky, M., 1975, "A framework for representing knowledge," *The Psychology of Computer Vision*, edited by P. H. Winston, McGraw-Hill, pp. 211–277.

Pople, H., 1982, "Heuristic methods for imposing structure on ill-structured problems," *Artificial Intelligence in Medicine*, edited by P. Szolovits, Westview Press, AAAS Symposium Series, pp. 119–190.

Shirley, M., Davis, R., Oct 1983, "Generating distinguishing tests from hierarchical models and symptom information," *Proc IEEE International Conference on Computer Design.*

Shirley, M., "Generating tests by exploiting designed behavior," *Proc AAAI-86.*

Shortliffe, E., Buchanan, B., 1975, "A model of inexact reasoning in medicine," *Mathematical Biosciences*, 23, pp. 351–379.

Williams, B., 1985, Principle designed based on qualitative behavioral descriptions, MS Thesis, MIT, Computer Science.

34

Silicon Compilation

Daniel D. Gajski, University of Illinois, Urbana, IL

The term silicon compilation was introduced by Dave Johannsen at Caltech (Johannsen, 1979), where he used it to describe the concept of assembling parameterized pieces of layout. This term has gained in popularity recently throughout the IC CAD community, where it has been used in a variety of different contexts.

In a narrow sense, it is an extension of the standard cell approach, where standard cells are replaced by parameterized cell compilers (Nance, 1983) that allow users to customize the cell functionality, electrical, and layout parameters. In the case of simple cells, such as NAND and NOR gates, a user can specify a number of inputs, choose among several drive buffers, and select the position of some I/O ports on the boundary of the cell. More recently, compilers for microarchitectural components—such as ROMs, RAMs, PLAs, and ALUs—were added to the basic SSI set. Because the number of options has increased with the complexity, special forms or menus have been provided for specification of compiler parameters.

In a much broader sense, silicon compilation can be defined as a translation process from a higher-level description into a layout. Here, a higher-level description defines a description that hides some level of detail from the user—but this description is not just a textual equivalent of the layout. This translation process can be broken into several steps, and each step can be considered to be a compiler for the lower level. In this way, we can define a logic compiler—which translates a description into a set of logic gates and flip-flops—or we can define a microarchitecture compiler, which translates an instruction-set description into a set of registers, buses, and ALUs.

In order to represent different approaches to silicon compilation, we will use the tripartite representation of design (Y-chart) shown in Figure 1 (Gajski, 1983). Multiple levels of abstraction or levels of detail are represented along each of the three axes. They represent three different domains of description: functional, structural, and geometrical. The levels of abstraction increase as one moves away from the vertex.

In the functional (or behavioral) domain we are interested in what a design does. We are not interested in how it is built. The design is treated as a black box with a specified set of inputs, a set of outputs, and a set of functions describing the behavior of each output as a function of the inputs and of time. For example, the Boolean expression "$x = a'b + ab'$ after 30 ns" [Figure 2(a)] indicates only the function of the cell whose inputs are a and b, and whose output x will become a logical one 30 ns after a and b become different, or logical zero 30 ns after a and b become the same. This expression does not say anything about the implementation or the structure of the cell.

In the more abstract levels of description—such as finite-state-machine and register-transfer description—time is replaced by the concept of state. On a higher level, such as an algorithmic level, the concept of state has been even further replaced by the concept of statement sequence, which only prescribes the order of statement execution. Several levels of abstraction can be identified in the functional domain (Walker, 1985). We use differential equations on the circuit level and Boolean expressions on the logic level. On the microarchitecture level, we use register-transfer (or finite-state-machine) description, which specifies for each control state the condition to be tested, all register transfers to be executed, and the next control state to be entered. An algorithmic description defines all the data structures and a sequence of transformations that manipulate them. At an algorithmic level, variables or data structures are not bound to registers or memories, and operations are not bound to any functional units or control state. The system or architectural description defines gross operational characteristics with performance specifications, without being concerned how data is manipulated or what algorithm is used.

A structural representation is the bridge between the functional and geometric representations. It is a one-to-many mapping of a functional representation onto a set of components and connections under constraints such as cost, area, and time. If, for example, the Boolean expression from Figure 2(a) is mapped onto a set of components consisting only of two-input NAND gates with a maximum delay of 10 ns, then one of the structural representations will consist of four NAND gates as shown in Figure 2(b). The structural representation does not specify any physical parameters, such as the position of the four NAND gates on a printed circuit board or on a silicon chip.

Sometimes the structural representation—such as logic or circuit schematic—may serve as a functional description. For example, the functional description obtained from the structure in Figure 2(b) would be $x = ((a(ab)')'(b(ab)')')'$. Such a derived functional description can be nicely used for design simulation and time verification, but it makes redesign with new design rules almost impossible.

Most commonly used levels of structural representation can be identified with the basic structural elements used. On the circuit level, the basic elements are transistors, resistors, and capacitors, while gates and flip-flops are on the logic level. ALUs, registers, RAMs, and ROMs can be used to represent register-transfer as well as algorithmic structures. However,

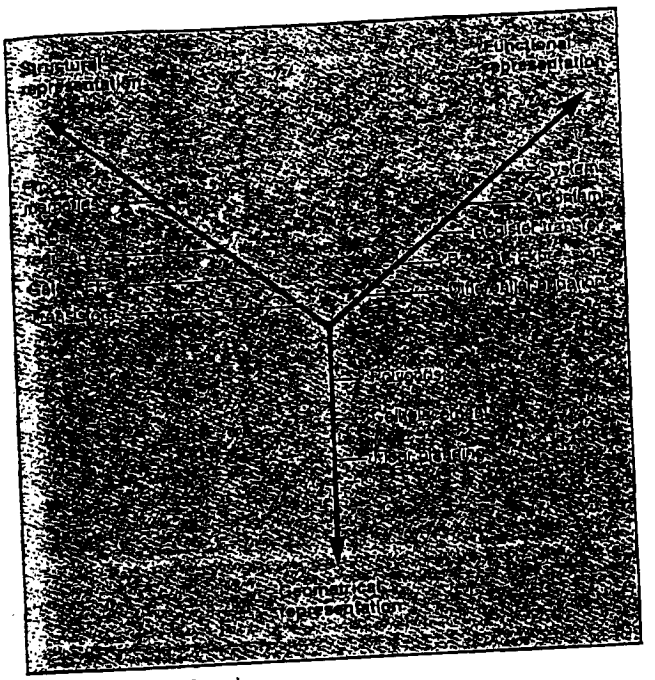


FIGURE 1. Y chart.

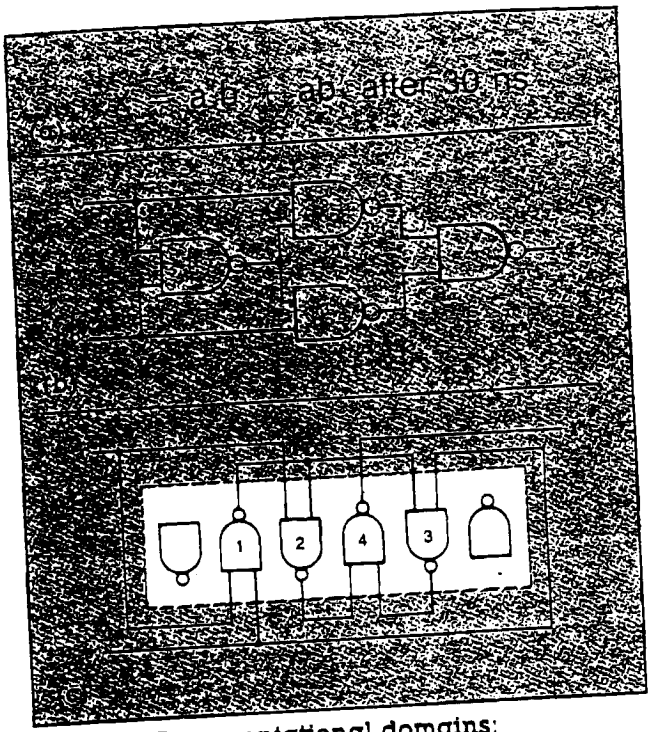


FIGURE 2. Representational domains: functional (a), structural (b), and geometrical (c).

at the algorithmic level, we tend to group microarchitectural components into data paths, control units, and data storage, and to be more concerned with synchronization and communication among the components than with their implementation. Processors, memories, and switches are used on the system level.

The geometrical representation ignores as much as possible what the design is supposed to do, and binds its structure in space (physical design) or to silicon (geometrical design). For example, if a gate array consisting of two-input NAND gates arranged in cells of six gates each is used, then a possible binding of the structure from Figure 2(b) is shown in Figure 2(c). Each NAND gate and each connection in the structure is assigned a physical location.

The structure-to-geometry mapping can be defined as a two-step process. The first step, usually called symbolic or topological layout, determines relative or approximate positions for all structural elements. The absolute positions are determined in the second step, after substitution of layouts for symbols and compaction. Although symbolic layout can be viewed as an independent design representation, we have included it in geometric representation in order to simplify our representation space. The most commonly used levels in geometric representation are mask geometries, cell placement, and floor planning with arbitrary-size blocks. Note again that the floor-planning level may cover several distinct levels in two other representational domains. On the system level, we may have physical partitioning and placement at the board and cabinet levels.

Silicon Compilation

The Y chart can be used to represent different VLSI design methodologies and pictorially explain differences among silicon-compiler-based design systems.

A gate-array or standard-cell methodology requires a designer to start with basic components such as gates, build a structure, define its function, and use it hierarchically to build higher-level structures. After the design is finished, it is flattened into the structure of basic components for simulation, placement, and routing [Figure 3(a)]. This methodology does not efficiently exploit the hierarchical nature of the design, because simulation, placement, and routing are performed on the lowest level of abstraction, where it is the most expensive. Furthermore, the fixed functionality, as well as the fixed electrical and layout properties of the basic components, lead to an inefficient layout.

Silicon-compilation methodology tries to overcome these deficiencies by providing basic components that can be fine-tuned to users' specifications on a higher level of abstraction, such as ALUs, PLAs, RAMs, data paths, controllers, and core microcomputers. Silicon compilers are programs that generate a layout description (geometric model), and possibly a corresponding structural description for simulation (simulation model) and timing analysis (timing model) when executed. The functionality, electrical, and layout properties for each basic component are passed as parameters to the corresponding silicon compiler. A hierarchical methodology is supported by allowing silicon compilers to call other silicon compilers.

As with gate-array and standard-cell methodologies, silicon compilation requires two experts. A silicon compiler writer constructs compilers for leaf cells, and then uses them to construct module compilers, processor compilers, and other high-level components. The task of a silicon compiler writer is represented by the outward-going spiral in Figure 3(b). A system or application designer specifies the design using a functional or structural description. In the former case, five different tasks must be performed before the design

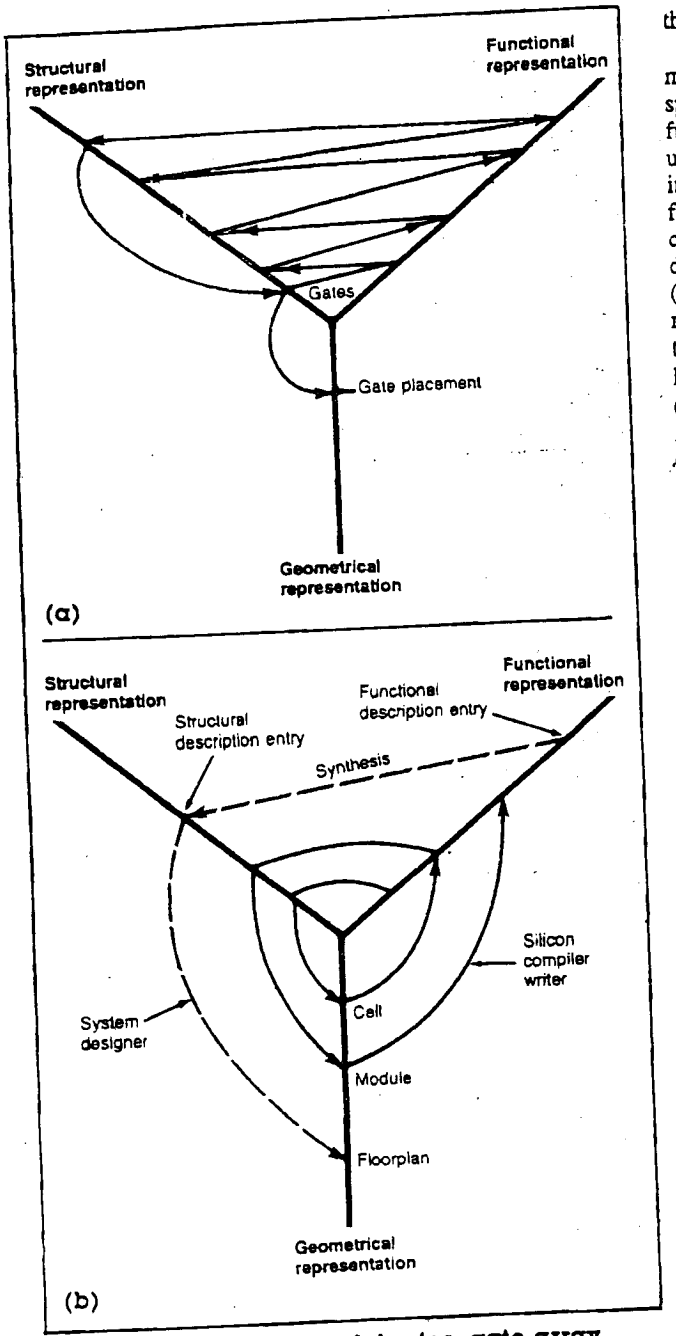


FIGURE 3. Design methodologies: gate array (a) and silicon compilation (b).

is ready for fabrication:

1. A functional description is translated into a structural description.
2. The layout of each of the structural components is instantiated by a silicon compiler.
3. All structural components are placed on silicon and routed.
4. Packaging is selected.
5. A test-vector set is generated.

In the latter case, the translation task is not needed, because the system designer specifies the design structure.

A typical IC design system based on silicon compilation methodology is shown in Figure 4. The design structure is specified by the user or generated by the synthesizer from the functional description. In either case, a menu/form package is used to capture the functional description of each component in the structure. The functional description is passed in the form of options or parameters to the corresponding silicon compiler. A technology file contains all process-relevant design rules used for generating a geometric model (layout). (In an interactive environment, layout or schematic editors may be used to alter compiler outputs. In this case, however, the "correct by construction" property of silicon compilers is lost). Similarly, timing, functional, or logic models are generated for each component in the structure. These models are linked together and passed to a timing analyzer or a simulator. Geometric models are linked together with placement and routing tools to form a chip composite. For interactive placement and routing, a composition editor or a silicon assembler can be used (Trimberger, 1984). A package editor is used to provide packaging information to the foundry. Similarly, the simulator provides a test set for testing the assembled IC.

Silicon Compiler System Characterization

In this section we will define basic characteristics for silicon-compiler-based IC design systems.

Market taxonomy. In Figure 3(b) we have seen that the world can be roughly divided into system designers, who use silicon compilers to design application-specific ICs, and compiler writers, who design silicon compilers to be used by the first group. The first group tends to be associated largely with system companies or divisions that do not have their own fabrication facilities, while the other group is associated mostly with semiconductor manufacturers that provide custom design and foundry services. This division in the marketplace is reflected in commercial silicon compilation systems. Silicon Design Labs Inc. (SDL) (Buric, 1985) offers a compiler-writing system, while VLSI Technology Inc. (VTI) (Nance, 1983) and Silicon Compilers, Inc. (SCI) (Johnson, 1984) provide silicon compilers for system designers. Seattle Silicon Technology Inc. (SST) provides both.

Hardware environment. Silicon compiler systems come in two flavors. They may be complete systems, with their own set of proprietary tools such as simulators, timing verifiers, and routers, for complete IC design. Or, they may be integrated into a standard CAE workstation, with most of the support tools being provided by the host workstation environment. The advantage of the first approach is possibly a more efficient database and interface between different tools, while the other approach offers a smooth transition from existing methodologies and CAE stations.

Portability. This characteristic describes the ease with which externally generated cells or modules can be integrated into a design generated by silicon compilers, and vice versa. Good portability assures smooth transition from manual methodology and saves design time by reusing already designed parts.

Input description. As we mentioned earlier, two different domains can be used for input description: structural and functional. For each domain, the input can be specified on one of several abstraction levels. Almost all commercial

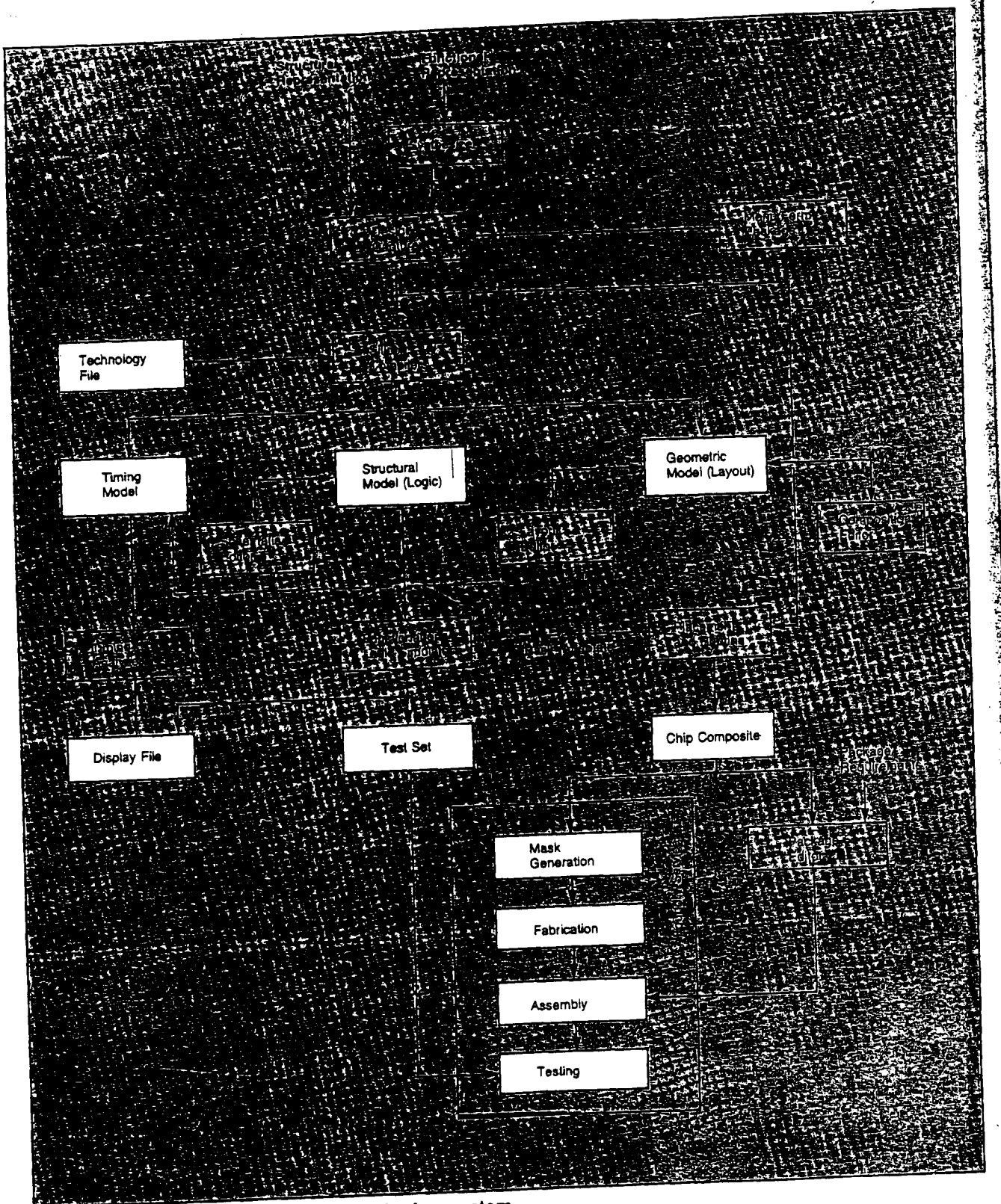


FIGURE 4. Silicon-compilation-based design system.

silicon compilation systems use the microarchitecture and/or logic structure as the input. A compiler-writing system such as GDS from Silicon Design Labs allows structures to be captured on any level, because a user is allowed to build custom compilers for his own structural components. The Metasyn silicon compiler from Metalogic uses a register-

transfer functional description that is automatically translated into control and data-path structures (Southard, 1983).

Compiler languages. Silicon compilers usually use two languages. The first language is used primarily for layout description. To make silicon compilers independent of design rules, layout description languages allow placement of components relative to each other or on a predefined virtual grid. At instantiation time, the layout is compacted according to design rules from a technology file. Some languages describe only layouts, while others are capable of describing several levels of abstraction. For example, the L language from Silicon Design Labs is capable of describing layout, circuit, and logic levels, which eases generation of different views of a design. Otherwise, a compiler must be run to generate a layout from which a circuit extractor can generate the circuit schematic. The second compiler language is used for describing simulation or timing models. This language can be a general programming language, such as C, which is used by SCI, SDL, and SST.

User interface. The user interface specifies ways the designer may interact with the system during the design phase, in addition to specifying the initial input description. Some systems allow layout or schematic editing after it has been generated by a silicon compiler. A layout editor is necessary during compiler writing, because a cell layout is at first created manually, then converted into text and parameterized. Some other systems allow preplacement and prerouting during design composition.

Module types. Module types can be divided into basic modules (such as SSI logic, registers, ALUs, PLAs, RAMs, ROMs, and data paths), whose operation takes one control state, and sequential modules (such as counters, finite-state machines, controllers, and processors), whose description encompasses more than one control state. VTI offers several types of controllers (interrupt, bus, DMA, and CRT controllers); SST offers a finite-state-machine generator; and SDL offers a core microcomputer.

Fabrication technology. Fabrication technology specifies the process for which a silicon compiler is written. Although silicon compilers can be made independent of design rules through a technology file, they are not process-independent—that is, the silicon compiler written for a CMOS process can not be easily derived from one for an nMOS process.

Foundry service. Some of the companies offering silicon compilers offer foundry services in-house, or they will provide the user with technology files for several selected foundries. The compiler-writing systems expect the process design rules for a particular foundry to be specified by the user.

Quality Measures. Three quality measures can be defined to evaluate silicon compilers:

1. Transistor density in mil²/transistor
2. Compilation speed in transistors/hour
3. Design time in transistors/person-hour

The above quality measures depend heavily on the complexity and the regularity of the design. Until standard benchmarks are established, these measures are just indicative of capabilities of silicon compilation and should not be used to compare different systems.

In addition to design functionality, the transistor density also depends on the fabrication process. SDL reported a density of 2.0 mil²/transistor for its core microcomputer (similar to the Intel 8051) in 3-μm CMOS; SST reported a density of 1.5 mil²/transistor for a 55,000-transistor design in 2-μm CMOS; while SCI has obtained 1.07–1.92 for different designs in 2-μm nMOS.

The density measure is similar to the computer performance measure of instructions per second, which is not very accurate, although it is a very popular approximation of performance. A better measure of density would use the design functionality as a normalizing factor instead of transistor count, because the existence of each transistor may be difficult to justify.

The compilation speed defines compiler run-time without plotting and should be normalized to the same machine. SCI reported compilation speeds from 5K–21K transistors/hour for different chips, while SDL reported 18K transistors/hour on the SDL 2000 core microcomputer.

The design time depends heavily on design complexity, as well as on the abstraction level of the compiler. SST experienced 6.1 transistors/hour, while SCI reported 8.2–25 transistors/hour for various designs. SDL obtained 420 transistors/hour by using a core compiler and adding random interface logic equivalent to 1,000 gates. Metalogic experienced 300 transistors/hour in a design of a 6502 processor. These results indicate an obvious trade-off between design time, compiler complexity, and high-level functional specification.

Design-Process Model

In this section we will try to define tasks that constitute a design process, and evaluate how these tools are supported in silicon compiler systems.

There are three basic tasks: refinement, optimization, and strategy selection. Refinement (sometimes called partitioning or decomposition) is a process of translating a behavioral or functional description into a structure of predefined components from the next lower level of design. This refinement is performed by a synthesizer and each of the silicon compilers shown in Figure 3. As we mentioned before, this translation is not unique; usually, several different styles can be selected. For example, the next state function in the control unit can be implemented as a one-way or two-way branch; multiple inputs to a functional unit can be implemented with a bus or a multiplexer; an adder can be designed as a ripple-carry adder or a carry-look-ahead adder; a chip may use a one-phase or two-phase clocking scheme; I/O ports on a cell can be on different sides; a cell layout may have different aspect ratios; a vertical or horizontal power architecture can be used in each cell. Most silicon compilers offer different output-buffer sizes; a variety of aspect ratios for RAMs and ROMs; different latching positions in data paths; direct, tristate, and precharged outputs; low-speed and high-speed PLAs; and so forth. Silicon Compilers Inc. offers two styles for testability—LSSD and Scanpath—and Seattle Silicon offers 1-phase and 2-phase clocking options.

The selection of styles depends on the goals assigned to a particular implementation. In order to automate the refinement, the process of translating overall design constraints into goals—and goals into different design styles—must be solved and captured in one form or another. Currently, a designer performs this translation using trade-off heuristics.

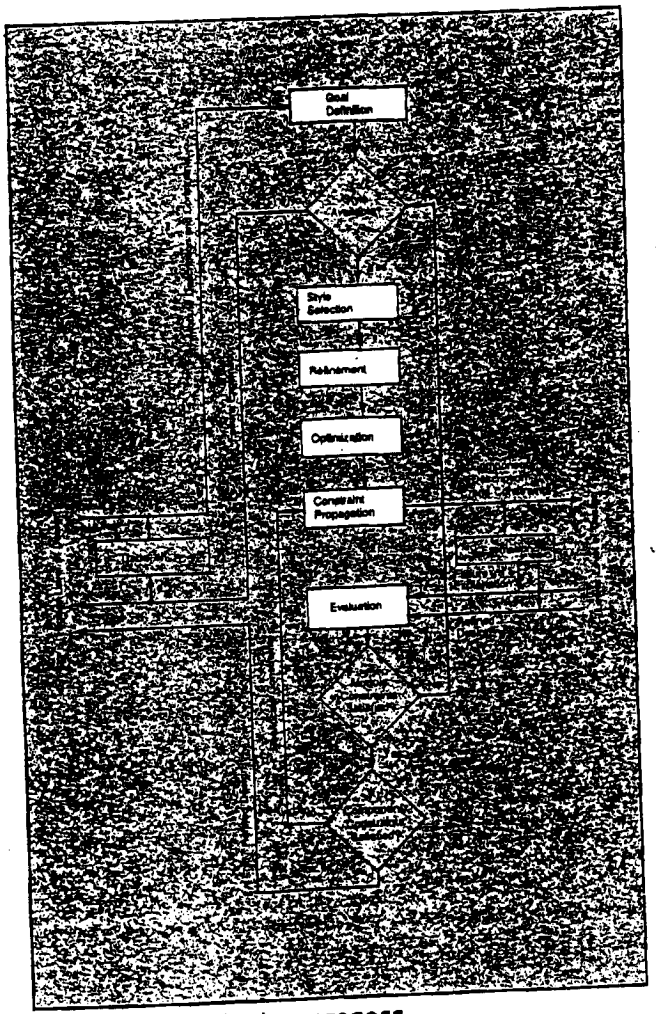


FIGURE 5. The design process.

An optimization step improves utilization of allocated resources, such as silicon area in PLA folding or layout compaction, number of tracks in channel routing, or number of functional units in a microarchitecture. An optimization can be performed after or together with the refinement. An optimization process can usually be defined by an algorithm. Compaction, placement, and routing optimizations are typical, and clock and power sizing and PLA folding are offered by some silicon compilers. The Metasyn compiler from Metalogic performs sharing or merging of functional units and buses on the microarchitectural level.

A strategy is a sequence of different refinement and optimization steps. The type and the order of steps in the sequence characterizes a strategy. Strategies are better understood at levels closer to physical design than on the higher, more abstract levels. It is natural, for example, that a symbolic layout be followed by geometric layout, followed by compaction afterwards. However, the order of register, unit, and bus allocation and their optimization on the register-transfer or microarchitecture level is not clear. Currently all silicon compilers allow only one fixed strategy.

The whole design process may be thought of as a set of refinements and optimizations at each level of design. In other words, it is a set of translations from a less-specific to a more-specific description of the design.

However, a design process includes more than just refinements and optimizations. At least three more mechanisms must be added. Constraint propagation processes the partition constraints assigned to a design module into constraints for each of its components. This task is presently performed by the designer. The second mechanism deals with evaluating the final design of each component, and estimating how well constraints have been satisfied—and finding the cause of the failure if they have not been satisfied. The evaluation process is supported by providing time analysis, cost, and power reports. For example, Silicon Compilers Inc. provides a clock report, I/O timing report, path delay report, violation report, and SCOAP report. The third task deals with performing a trade-off analysis in the event that there is no design that satisfies given constraints. The trade-offs are made by choosing a different design style, such as choosing a ripple-carry adder over a carry-look-ahead adder if the smaller-area goal is more important than the higher-speed goal.

A design-process model including constraint propagation, design evaluation, and style selection is shown in Figure 5 (Gajski, 1985). For each design level and for each module, design constraints assigned at a higher level of design are translated into an ordered list of goals. A certain style of design is chosen according to the goal list. If no style satisfies the given goals, a failure explanation, which focuses on the part of the design and the reason why it failed to satisfy constraints, is reported to the higher level of design. After a design style is selected, the module description is refined and possibly optimized. A refinement generates a structure of components—each of them to be refined on the next lower level of design. Constraints for each component are derived from the module constraints and passed to the next level of design together with the behavioral description of each component. After a module description is completely refined, the resulting design is evaluated. If module constraints are violated, a new style is selected. On the other hand, if module constraints are satisfied, but some of the module's components have failed to meet the constraints, then a new assignment of constraints is attempted.

It is not necessary to completely refine a design in order to find that its present style is not satisfactory. Design estimators can be used to replace a long refinement process. They are much simpler, faster, and obviously less accurate than performing the real refinements and optimization. Hence they are usually used during preliminary design or during a feasibility study, when the speed of searching the design space is more important than the quality of the design. Usually, routing, area, power, size, and performance estimates for the composite design or for each module are offered. Similarly, a design failure does not have to be propagated to the higher level of abstraction for constraint relaxation. Trade-off analysis can be performed on any level in order to avoid extensive backtracking if trade-off guidelines are provided.

Currently, goal definition, style selection, constraint propagation, design evaluation, and trade-offs are made by a designer with the help of CAD tools, such as simulators and timing verifiers, for example.

Conclusion

In this brief article we have presented silicon compilation

as a new and evolving methodology that makes custom silicon affordable and the design-time bottleneck deferable to the distant future. Silicon compilation is an evolutionary technology that manages VLSI complexity by using hierarchical methodology on higher levels of abstraction. It blends well with present CAD tools, so we may expect more silicon compilation systems running on standard CAE stations to appear in the near future.

Present trends are toward more-powerful layout and structure description languages, integrated databases for easy conversion between levels of abstraction, and the merging of compiler-writing systems with application-oriented design systems. Future trends include functional description input, automatic synthesis tools, and AI-based tools for complete design-process automation. □

Acknowledgments

I would like to thank Clam Meas and Gary De Palma of Silicon Compilers Inc., Vince Corbin of Seattle Silicon Technology, Misha Buric and Tom Matheson of Silicon Design Labs, and Jay Southard of Metalogic for providing data and insightful comments during the writing of this paper.

References

Buric, M.R., and T.G. Matheson. 1985. "Silicon Compilation Environments," *Custom Integrated Circuits Conference*, Portland, OR.

Gajski, D.D. June 1985. "ARSENIC Silicon Compiler," *International Symposium on Circuits and Systems.*

Gajski, D.D., and R.H. Kuhn. December 1983. "New VLSI Tools," *IEEE Computer.*

Johannsen, D. 1979. "Bristle Blocks: A Silicon Compiler," *16th Design Automation Conference*, San Diego, CA.

Johnson, S.C., and S. Mazor. October 1984. "Silicon Compilation Lets System Makers Design Their Own VLSI Chips," *Electronic Design.*

Nance, S., C. Starr, B. Duyn, and M. Kliment. September 15, 1983. "Cell-Layout Compilers Simplify Custom IC Design," *EDN.*

Southard, J.R. December 1983. "MacPitts: An Approach to Silicon Compilation," *IEEE Computer.*

Trimberger, S. 1984. "VTIcompose—A Powerful Graphical Chip Assembly Tool," *International Conference on Computer-Aided Design*, Santa Clara, CA.

Walker, R.A., and D.E. Thomas. 1985. "A Model for Design Representation and Synthesis," *22nd Design Automation Conference*, Las Vegas, NV.

About the Author

Daniel D. Gajski is an associate professor in the computer science department at the University of Illinois at Urbana-Champaign. Prior to joining the university in 1978, he had 10 years of industrial experience in areas such as digital circuits, switching systems, supercomputer design, and VLSI structures. Gajski received the Dipl. Ing. and the M.S. degree in electrical engineering from the University of Zagreb, Yugoslavia, and the Ph.D. in computer and information sciences from the University of Pennsylvania.

DEF076341

35

United States Patent [19]

Bryant et al.

[11] **Patent Number: 4,638,442**

[45] **Date of Patent: Jan. 20, 1987**

[54] **COMPUTER AIDED DESIGN METHOD AND APPARATUS COMPRISING MEANS FOR AUTOMATICALLY GENERATING PIN-TO-PIN INTERCONNECTION LISTS BETWEEN RESPECTIVE DISCRETE ELECTRICAL COMPONENT CIRCUITS**

[75] Inventors: **Stewart F. Bryant**, Redhill; **Stephen J. Baker**, Brighton; **Richard A. Cook**, Haywards Heath, all of England

[73] Assignee: **U.S. Philips Corporation**, New York, N.Y.

[21] Appl. No.: **669,207**

[22] Filed: **Nov. 6, 1984**

[30] **Foreign Application Priority Data**

Nov. 9, 1983 [GB] United Kingdom 8329888

[51] **Int. Cl.**[4] **G06F 15/60**

[52] **U.S. Cl.** **364/489**

[58] **Field of Search** 364/488, 489, 490

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,567,914 3/1971 Neese et al. ..... 364/489 X

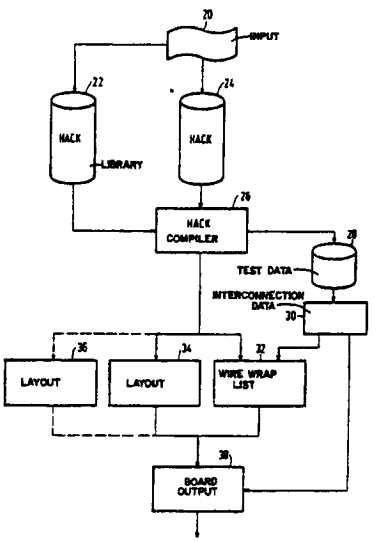
Primary Examiner—Michael L. Gellner

Attorney, Agent, or Firm—Jack E. Haken; James J. Cannon, Jr.

[57] **ABSTRACT**

Computer aided design method and apparatus which automatically generate pin-to-pin interconnection lists between respective discrete electrical component circuits. Input devices provide for entering a list of components used, a description of those components and the circuit interconnection description. The memory store contains a library of standard component descriptions. A processing unit process the entered circuit description and selected component descriptions from storage to produce an interconnection list in the form of an individual component pin to individual component pin connection.

9 Claims, 3 Drawing Figures



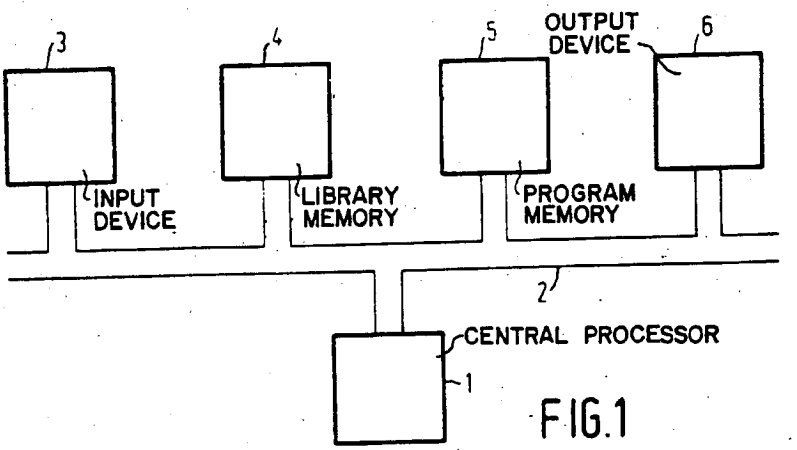

3
4
5
OUTPUT DEVICE
6
INPUT DEVICE
LIBRARY MEMORY
PROGRAM MEMORY
2
CENTRAL PROCESSOR
1

FIG.1

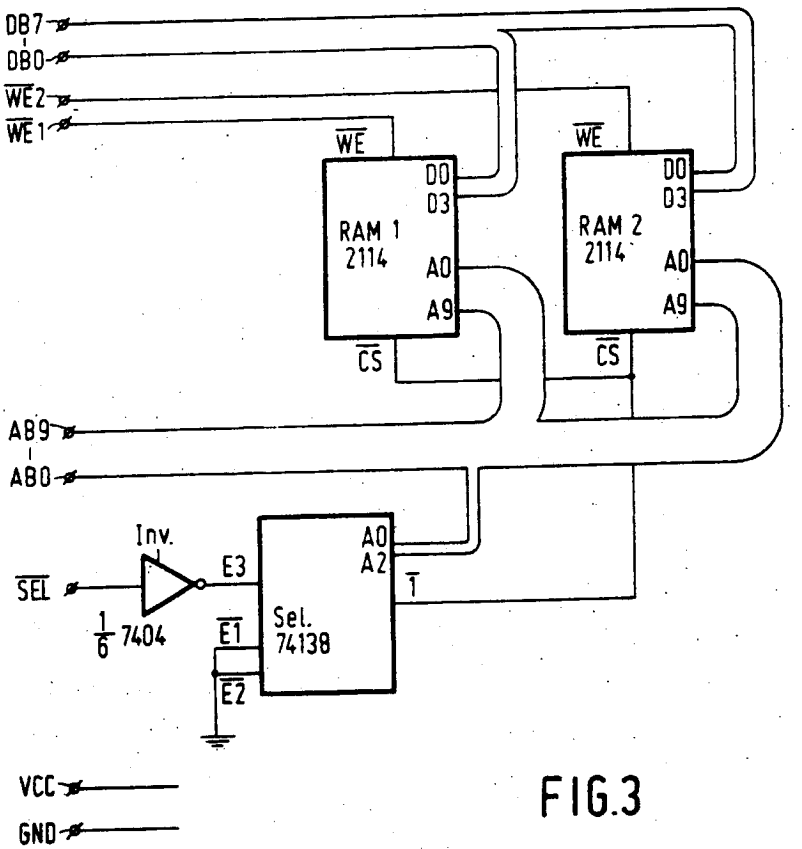

DB7
DB0
WE2
WE1
WE
DO
D3
RAM 1 2114
A0
A9
CS
WE
DO
D3
RAM 2 2114
A0
A9
CS
AB9
AB0
Inv.
SEL
1/6 7404
E3
E1
E2
Sel. 74138
A0
A2
Y
VCC
GND

FIG.3

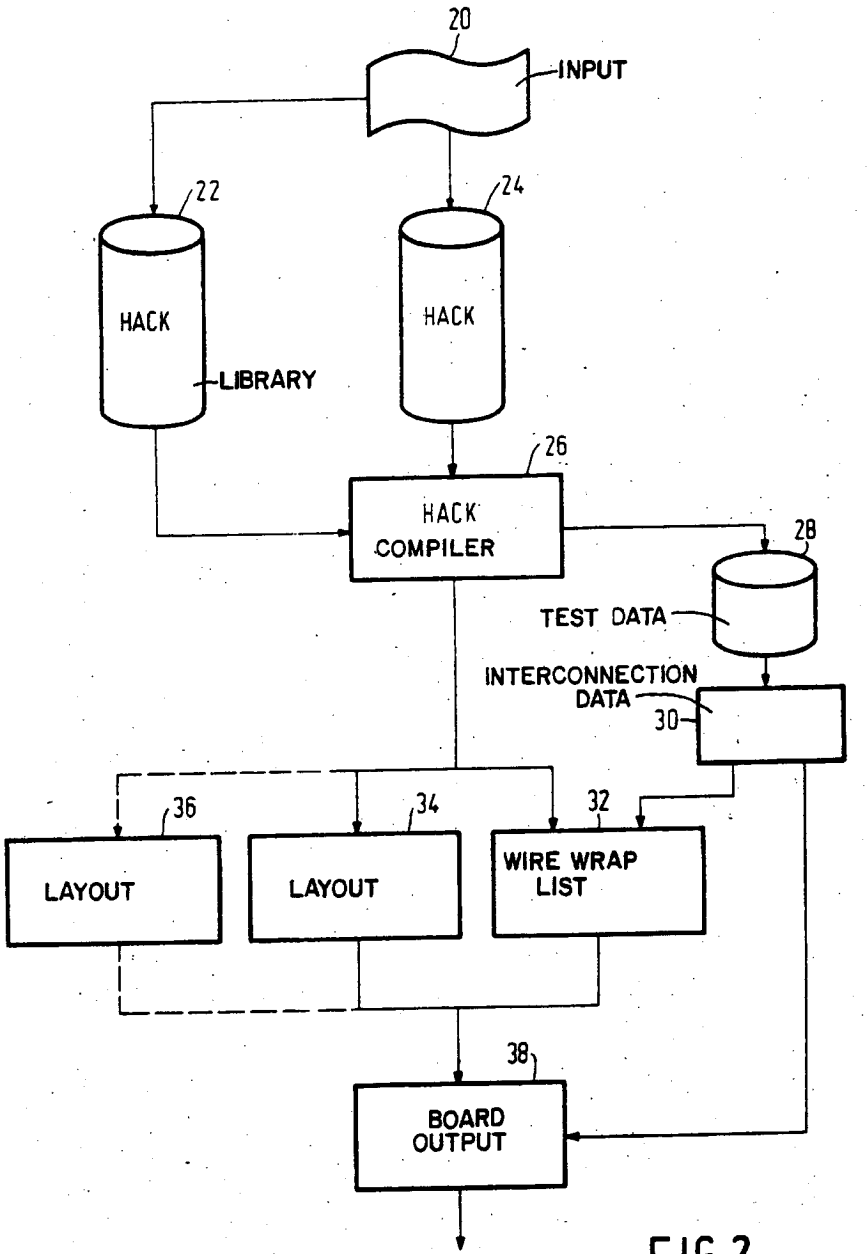

20
INPUT
22
24
HACK
LIBRARY
HACK
26
HACK
COMPILER
28
TEST DATA
INTERCONNECTION
DATA
30
36
34
32
LAYOUT
LAYOUT
WIRE WRAP
LIST
38
BOARD
OUTPUT

FIG.2

COMPUTER AIDED DESIGN METHOD AND APPARATUS COMPRISING MEANS FOR AUTOMATICALLY GENERATING PIN-TO-PIN INTERCONNECTION LISTS BETWEEN RESPECTIVE DISCRETE ELECTRICAL COMPONENT CIRCUITS

BACKGROUND OF THE INVENTION

The invention relates to a computer aided design method and apparatus comprising means for automatically generating pin-to-pin interconnection lists between respective discrete electrical component circuits. An example of a computer aided design/assembly package is the RACAL REDAC Mini/Maxi PCB Design System, described in the Data Preparation Manual by the Technical Writing Group from the manufacturer, RACAL REDAC Ltd., Newton, Tewkesbury, Gloucestershire, GL 20 8HE, England.

OBJECT AND GENERAL ASPECTS

The invention is particularly applicable in the design of printed circuit boards, although the technology of the interconnection between the discrete circuits may be effected by electrical connections, fiber optics or other ways. The problem addressed by the invention occurs when the electrical circuit has been drawn on the level of discrete elements, ranging from transistors, resistors and capacitors, via gates, registers, and the like, and sometimes up to VLSI microprocessors. Thereafter, the realization must be effected from a limited selection of standard circuits or modules. The output of the above-mentioned apparatus may be used directly in a further machine stage of a computer aided design system. Similar conditions apply when producing interconnection lists for wire wrapped circuit boards where the lists are used be persons performing the wire wrapping or are fed to an automatic wire wrapping machine.

The present state of the art is exemplified by an article HISDL-A structure description language; by W. Y. P. Lim, Communications of the ACM, Vol. 25, No. 11, November 1982, pages 823–830. The inventors of the present invention have realized that the inputting of the circuit interconnection description should be made as straightforward as possible. One of the improvements therefore is the provision for arrays of named components; in practice certain components, notably memory components are used array-wise to enlarge the circuit's capacity. The invention provides a solution in that it provides an arrangement for generating a component interconnection list for an electrical circuit, each interconnection being in the form of a link between a first individual component pin and a second individual component pin, said arrangement comprising means for entering

- a component list containing any component used,
- a component description of any component used inclusive of a functional pin description thereof, and
- a circuit interconnection description containing elements of the form of a first array of pins of a second array of named components connected to a third array of pins of a fourth array of named components, at least one array being non-unitary,

wherein for any circuit interconnection description element in which both said first and said third arrays are non-unitary the first and third arrays have equal numbers of array elements, said arrangement furthermore

containing library means for storing said component descriptions as standard component descriptions, processing means for processing, under program control, the circuit interconnection description and the list of components used while addressing said library means for accessing said component descriptions, and output means for after said processing outputting said component interconnection list.

Herein, a unitary array is defined as an array having only a single element. Thus, the invention provides an automatic facility for interconnecting arrays, no specifying of a loop being necessary. The present realization is particularly advantageous because bus-type interconnections are usually specified on a 1-to-1 basis inclusive an offset quantity. The present invention is specifically data-oriented. The interconnection description may be in the form of:

X[a:b].A[c:d], Y[e:f].B[g:h];

where

X and Y are component pin function descriptions,

A and B are component means,

a, b, c, d, e, f, g, and h are integers or letters which may take any value provided that the difference between a and b is equal to the difference between e and f unless either of those differences is equal to zero in which case the other may take any value, the dot (.) represents "belonging to", and the comma represents "is connected to".

A frequent requirement is that of connecting a number of components to an n-bit address or data bus, where n is an arbitrary number but typically eight or sixteen. The components may be, for example memory devices, microprocessors, registers, buffers, etc. Different manufacturers of these devices use different conventions to define the significance of each bit on the address or data bus. Thus for example in an 8 bit data one manufacturer may designate the most significant bit D0 and the least significant D7 while another manufacturer reverses this convention and uses D7 for the most significant bit. If a circuit is designed using components from two different manufacturers a cross connection has to be specified and this frequently gives a rise to errors in specifying the connections. The arrangement according to the invention includes a store holding a library of component descriptions which can be accessed by a computer and into which component descriptions can be inserted and from which previously stored descriptions can be accessed. In one form the invention enables the use of a simpler interconnection description by allowing the entry of a component interconnection description in the form

D[1:n].A, D[n:1].B; (1)

This description is interpreted by the arrangement as an instruction to produce an interconnection list containing instructions to connect the pin designated D1 of a component A to the pin designated Dn of the component B, the pin designated D2 of a component A to the pin designated D(n−1) of component B and so on until pin Dn of component A is connected to pin D1 of component B. Thus the possibility of generating erroneous connection information is reduced provided that the correct component descriptions have been entered into the component library. The input description may alternatively, if appropriate for the particular components, be

D[1:n].A, D[1:n].B; (2)

which is interpreted in a similar manner, the difference being that pins on component A are connected to corresponding pins on component B. Further the description

D[1:n].A, D[m:n+m].B; (3)

is interpreted as connecting n pins of component A to n pins of component B on a one to one basis but with the pin numbers offset by m. It is not permissible to specify a different total number of pins on Component A with respect to Component B.

It may also or alternatively be desired to connect a plurality of pins on component A to a single pin on component B or vice versa. In a further form the invention enables the use of a simple interconnection description by allowing the entry of a component interconnection description in the form

X[1:n].A, Y.B; (4)

This description would be interpreted by the arrangement as an instruction to produce an interconnection list containing instructions to connect n pins designated X on component A to the pin designated Y on component B. This is the only condition under which unequal numbers of pins to be connected on the two components are allowed.

A further interconnection requirement which frequently occurs is to connect pins of n components of type A to pins of m components of type B where n and m are integers which may be equal or unequal. In a still further form the invention enables the use of a simple interconnection description by allowing the entry of a component interconnection description in the form

X.A[1:n], Y.B[1:m]; (5)

This description is interpreted by the arrangement as an instruction to produce an interconnection list containing instructions to connect pin X on n components of type A together and to pin Y on m components of type B.

The description (5) may be combined with any of the descriptions (1) to (4) and the combined description will be interpreted accordingly by the arrangement and the appropriate interconnection list generated. The above explains the interconnection of two arrays of components; in similar way the chaining three or more arrays of components may be expressed by concatenation of the expressions herebefore, such as:

X[a:b].A[c:d], Y[e:f].B[g:h], Z[i:j].C[k:l];

The invention further provides a method of generating component interconnection lists for an electrical circuit in the form of an individual component pin to individual component pin instruction comprising: forming a circuit interconnection description of the form

X[a:b].A[c:d], Y[e:f].B[g:h];

where

X and Y are component pin function descriptions,

A and B are component names,

a, b, c, d, e, f, g, and h are integers which may take any value provided that the difference between a and b is equal to the difference between e and f unless either of those differences is equal to zero in which case the other may take any value,

and the comma represents "is connected to";

forming a list of components used and a description of those components; storing a library of standard components descriptions; and processing the circuit description and selected component descriptions from the library to produce an interconnection list in the form of an individual component pin to individual component pin connection.

The method may further include the step of declaring all components and named signals prior to processing the circuit description.

BRIEF DESCRIPTION OF THE FIGURES

An embodiment of the invention will now be described, by way of example, with reference to the accompanying drawings, in which:

FIG. 1 shows in block schematic form an arrangement for generating component interconnection lists according to the invention;

FIG. 2 shows the process of generating a circuit board layout;

FIG. 3 shows a circuit diagram of an electrical circuit for which a component interconnection list is to be generated.

DESCRIPTION OF AN EMBODIMENT OF THE ARRANGEMENT

FIG. 1 shows an arrangement for generating lists of component interconnections comprising a central processing unit 1 which is linked via a bus 2 to input means 3, a library 4, a program store 5, and output means 6. The input means may take any convenient form, for example a keyboard or a paper tape reader. The output means may take many forms depending on the use to which the information is to be put, for example a printer where the interconnection is to be made manually such as by wire wrapping or where the information is to be manually introduced into a computer aided circuit layout package. Alternatively the information could be transferred directly to a computer aided circuit layout package over a data link or could be transferred using any suitable intermediate storage medium such as paper or magnetic tape. The library 4 is a store containing details of the components used i.e. size, shape, number of pins and functions of each terminal or pin. The program store 5 contains instructions for the central processing unit 1 to enable the required output information to be generated from the input information. Although the arrangement in FIG. 1 is shown as a single-purpose arrangement; certain other tasks could also be executable thereon, however.

FIG. 2 shows, in the form of an arrangement of modules the process of generating a circuit board layout; reference is still be had to FIG. 1. Now, a suite of programs stored in program memory 5 is used to enable the arrangement to generate lists of component interconnections. Input means 20 contains component descriptions which are forwarded to module 22 and connection specifications which are forwarded to module 24. Module 26 is fed by both modules 22 and 24. In module 26, one of the programs, called CHOP enables the designer to rapidly convert a circuit diagram (fed from module 20) into a form compatible with the input format of a commercially available printed circuit board (PCB) design system. A second program, called "Trace" as-

sists in the construction of wire-wrapped prototypes and the debugging of the finished card.

Using the system, designated HACK (Hardware Compilation Kit) the designer is able to describe the circuit interconnection in a high level language syntactically similar to the C programming language. The input language allows component reference by functional name, and includes facilities to simplify the description of repetitive component and interconnection circuit structures, such as arrays and buses. The library of standard components is available in module 22, the source code of the HACK language is present in module 24. The function of module 26 is that of a compiler; generally compilers are well-known in the art of data processing. If necessary, the designer may update the library (module 22). The compiler (module 26) generates a file of pin connections that may be input into the further modules. In this respect, module 34 represents the commercially available REDAC (see hereabove) printed circuit board layout service.

To aid in the debugging of boards copiler module 26 also produces a list of any pins the designer may have left unconnected; also, a pin diagram of all discrete parts referenced in the source code is made available for display. This module furthermore generates a connection tally file which shows the number of connections between various pairs of components. Especially in a computer aided design system this may be used in placing the discrete elements. Those pairs having a high number of interconnections are by preference placed close together.

Module 28 is addressed by module 26 and contains test data. For example, in case of a memory component here a number of test patterns of known "difficult" character are stored.

The HACK input language is block-structured and in many respects a hybrid of the popular "C" and Pascal programming languages. These languages have been well-published and are available in the art of programming. The overall programming is like the language C, with the use of curly brace characters "{}" as block delimiters and C-like preprocessor commands and comments. The one principle deviation from this style is the use of a Pascal-like "with" construct to simplify the description of circuit areas that contain frequent reference to a particular component or component array.

Module 30 receives the test data from module 28; furthermore, this module (TRACE) traces all interconnections made for a particular board. It produces an output file for assisting the wire-wrapping. Module 32 is addressed by modules 26 and 30 and generates the instructions for wire-wrapping in the form of lists of items (from-to connections). Alternatively, module 26 may address module 34 containing the RACAL REDAC pcb layout service package referenced hereabove. Alternatively, another layout service package symbolized by module 36 could find application. Outputs of modules 34, 36 would also be in the form of from-to interconnection lists, where the actual interconnection could be made by hand or automatically.

Module 38 symbolizes the actual board manufacture operation as governed by the connection instructions from either of modules 32, 34, 36, or furthermore the test operation under control by test generating module 30. The output of module 38 is a series of finished boards, either accepted or rejected. The general idea of FIG. 2 is to show the interconnection of the respective process modules: the routing of the boards during execution of module 36 is not shown. Neither are shown the respective output formats of data such as video displays, hard copies by printers or control tapes either punched or magnetic.

DETAILED DESCRIPTION OF THE REALIZATION

The HACK description of a circuit starts with a header of the form:

```
board Example [euro-card]
```

giving the board the name "Example" and describing to a PCB design system, such as the RACAL REDAC Mini/Maxi PCB Design System, the board's physical dimensions by reference to a layout library description contained in the library 4. This is followed by the component and interconnection descriptions, enclosed within a pair of curly braces. Instructions for labelling the final printed board with its name and date of compilation man be automatically generated. Three types of block are permissible in describing the circuit: component description blocks, library blocks, and interconnection blocks.

Component description blocks relate a generic component type to the physical shape and pinout discriptions stored in the layout system's library. It also describes the mapping of the symbolic pin descriptions onto the actual pin numbers. This is analogous to the definition of a data structure in the C programming language—thus:

```
struct 2114
/*
1024 × 4 bit Static RAM.
*/
{
shape = L3004;
1 = A6 ; 18 = +5v ;
2 = A5 ; 17 = A7 ;
3 = A4 ; 16 = A8 ;
4 = A3 ; 15 = A9 ;
5 = A0 ; 14 = D0 ;
6 = A1 ; 13 = D1 ;
7 = A2 ; 12 = D2 ;
8 = CS/ ; 11 = D3 ;
9 = GND ; 10 = WE/ ;
}
```

Herein, 2114 is a type number to INTEL Corporation. L3004 is a shape name. The input language is completely free format, statements are terminated by a ";" character. Comments are contained within a "/*" "*/" pair and may be inserted at any point within the code (except within identifiers and other comments), and are replicated in the PCB layout description so that they may be used to identify particular interconnection blocks, or give layout staff special instructions.

All components must be declared prior to use. The process of component declaration is used to assign the component a functional name relevant to its use in the design thus:

```
struct elem inverter = 7404;
```

This declares one component of type "7404" and gives it the name "inverter"; herein, 7404 is a type number in the well-known series of TTL IC components to Texas Instruments and other sources. A more advanced component declaration is:

```
struct elem video_RAM[1:4]=2114;
```

which defines an array of four video RAMs (Random Access Memories) called RAM1, RAM2, RAM3 and RAM4 each of which is of type 2114. Arrays need not only consist of numbers:

```
struct elem video_RAM[A:D]=2114;
```

would also create an array of four RAMS called RAMA, RAMB, RAMC and RAMD.

Alternatively the declaration may be combined with the generic description, such as:

```
struct Electrolytic_Capacitor
{
shape = L3546;
1 = + ;
2 = − ;
}smoothing_capacitor ;
```

To avoid the repeated description of commonly used components, with the attendant risk of error, reference to one or more libraries is permitted in the component declaration:

```
library 74_series_TTL
{
elem selector = 74138 ;
elem inverter = 7404 ;
}
```

libraries contain component descriptions identical to those shown hereabove, and may be compiled separately.

Circuit interconnections are described in interconnection blocks delimited by curly braces. The connections within a block are presented to the layout package as a block to enable the distinction of different types of signal, for example signals, power, ground, buses, and varying track widths. The first statement of a connection block may be a track width specifier, which determines the track width used for the remainder of the block. For example

```
trackwidth=7;
```

The trackwidth statement may omitted, in which case the compiler picks a sensible default value.

A pin is referenced by means of a pin name and a component name, separated by a dot. The identifier:

```
Q1/.flip_flop
```

therefore refers to the Q1/pin of a previously declared flip-flop package. Note that the use of the slash character to denote active low signals is a convenient convention for peripherals that do not support overscoring. Interconnections are made by listing pin identifiers as follows:

```
Q1.inverter, E3.selector;
```

This causes the compiler to generate output to connect the Q1 pin of the inverter to the E3 pin of the selector. The list may contain any number of chained interconnections and may be continued on as many lines as required, a semicolon terminating the chain. A1.inverter, E3.selector, D1.flip_flop;

An array of pins may be connected automatically by the use of the pin range notation, thus

```
A[0:9].RAM, Q[1:10].buffer;
```

which would connect the 10 address pins of the device called RAM to the 10 output lines of the device called buffer, on a one to one basis. This dramatically reduces the amount of effort required to specify bus interconnection. The pin range notation may also be used to specify a multipin to single pin interconnection of the form:

```
D[1:5].buffer, Q1.inverter;
```

which would connect all 5 buffer input pins to the single output pin of the inverter. In the case of unequal connection ranges, with one of the ranges not equal to one, a compiler error message is generated.

The use of range notation may equally be applied to the interconnection of component arrays, and indeed the two range formats may be combined:

```
A[0:9].video_RAM[1:4], Q[1:10].buffer;
```

This statement would connect the RAM address pins to the buffer output pins on a one to one basis for the whole RAM array. One normally error prone task is the interconnection of two signal buses with the connection order of one inverted—this is simply achieved by the use of the pin range notation with the pin range order reversed in one of the cases.

```
A[0:4].RAM, D[4:0]. buffer;
```

This example would connect address 0 of RAM to input 4 of buffer, etcetera.

A second multiple component specification is the asterisk notation, in which an "*" character may be use in the place of one or more component names to mean "all applicable components". Usually, this asterisk is then preceded by a dot. The code +5 v.Connector_J3, +.smoothing_capacitor, +5 v.*; would be sufficient to specify the interconnection of the +5 volt supply for the entire card.

The inclusion of a "with" construct in the HACK input language simplifies the circuit specification when a large number of connections are being made to a particular component or component array. The semantics of this statement are similar to that employed in the Pascal programming language. Throughout the following block, any pin without a component name will be treated as belonging to the component (or component array) specified in the "with" clause. Codes of the form

```
with RAM[1:4]
{
Q[0:9].buffer, A[0:9];
CS/ , 1/.selector ;
{
```

would create interconnections making the assumption that pins A[0:9] and CS/ belonged to the array of four devices called RAM[1:4]. The "with" statement can be a powerful structuring tool as designers tend to work around the board generating "with" statements for each major component in the circuit. Nested "with" statements are not permitted.

It is frequently the case that the designer wishes to assign to a signal a logical name, and arrange to connect the pins of various components to that named signal. Another familiar problem is that of interconnecting the boundary signals arising through the partitioning of a circuit diagram over a number of sheets of paper. HACK provides a solution to this problem by the use of so called "external" signals. These are denoted in the circuit description by prefixing the signal identifier with a tilde ("~"). Pins are connected to the external signals in the conventional manner:

```
A1.RAM1,~address1 ;
.
.
~address1, Q1.buffer ;
```

The compiler is able to resolve any previously omitted connections at link time by generating a list of external signal interconnections. Prior to use, these signals must be declared by the use of an extern statement:

```
extern
address[0:7], GND, VCC ;
```

Note that the usual pin range construct is equally valid for externals.

To permit the use of symbolic constants in the HACK circuit description, and to enable the designer to split the design over a number of files, a C-like preprocessor is provided. Two types of command may be issued to the preprocessor: ≠define causes the preprocessor to perform a textual substitution of one symbol for a string of symbols.

```
≠define RAM_ARRAY 0:4
struct elem video_RAM [RAM_ARRAY] = 2114 ;
```

This has the effect of replacing the symbol "RAM_ARRAY" with the string "0:4" throughout the remainder of the code. The substitution text can be any sequence of characters up to the end of the line. This instruction may be useful where a particular group or array of components is frequently referred to and an alteration is made to the number of components in the group or array. In that case only the ≠define instruction has to be amended the corresponding changes being made automatically throughout the program.

The other preprocessor directive is ≠include, which causes the specified file to be included into the main program at the specified point.

```
≠include "videocircuit"
enable.RAM [RAMARRAY],~GND ;
```

To illustrate the use of the language in the discription of a design, a simple circuit diagram is shown in FIG. 3, and its corresponding HACK program is given in Appendix 1. Since many of the examples used in this description are taken from the Appendix the relationship between the conventional circuit diagram and its HACK description will be clear.

HACK is capable of producing a number of useful output files in addition to the one containing the code required as input to the PCB layout package. In the event of an error in the source code an error listing can be produced, which may be either a short summary of the errors, or a more traditional program style listing showing the errors in context.

A list of unconnected pins may be produced to assist the designer in determining the completeness of the circuit description and references the unconnected pin by its true component number and pin number as well as by its HACK name as used in the source code may be made. The component generic type may also indicated. The component number is a number allocated to each component by HACK for reference on the PCB, circuit, and other documentation. This numbering conforms to the traditional numbering style (i.e. IC1, TR1, R22), but the styles can be changed if necessary.

A component list can be produced for use in the fabrication and documentation of the board and a "tally" file may be produced showing the number of connections between each component and all other components in the circuit. The latter is of particular use during the construction of prototype circuits, where it is of assistance in the placement of the components.

A picture file can be produced showing the pinout of all the integrated circuits used (shown from both sides of the board) for use in debugging. The compiler can also make available an intermediate code file showing the generated interconnections, alongside the expanded source code that generated it. This is used by the trace utility, but may also be of use to other post processing utilities. The final file produced contains the HACK output used in the actual generation of the board.

Part of the code generated by the compiler from the source code in Appendix 1 for a Racal REDAC PCB layout systems is shown in Appendix 2. As an alternative, the user may request the code generator in HACK to produce a series of wire-wrap instructions for a wireperson to be used in the construction of a prototype circuit. These instructions list the connections in terms of component and pin numbers, and also suggest a suitable wire colour. Modified versions of HACK may be used to generate code for other layout systems.

Trace is a utility used in conjunction with the HACK system to trace connections. Trace examines the HACK intermediate code file and enables the designer to display interactively the physical interconnections produced from the HACK code. The program will tell the designer which pins of what other components are connected to a given pin, understanding component number, pin number and HACK symbolic descriptions. In order to assist in the identification of the interconnection on wire-wrapped prototype boards, the wire colour given in the wireperson's instructions is also shown. This makes life much easier when the designer and constructor are not the same person.

HACK is written in the C programming language and can be inplemented, for example under the CP/M (in BDS-C for the 8080/Z80), UNIX, and VAX/VMS operating systems. Porting the package to other operating systems would be a relatively simple exercise given the availability of an adequate C compiler. The object code of the largest overlay of the HACK compiler occupies about 60K bytes on an ONXK C8002 UNIX system. The restrictions that UNIX V7 place on the size of an image's data space limits the size of board that can be compiled. It is anticipated that a future switch to the UNIX System III operating system, or the recompilation of the program suite with the VAX11-C compiler on a VAX11/780 will result in a version of the compiler that places no reasonable restriction on the size of circuit that can be compiled.

It should be noted that throughout the description and Claims the symbols use are purely arbitrary and other symbols could be used depending on those available on the precise apparatus used, e.g. input keyboard and output printer. It would be obvious that other languages or other constructs could be used to realize the invention. The following advantageous points of the invention are summarized:

- there is an automatic array connect construct; especially when bus structures are of a one-to-one plus offset nature this is an elementary solution. Notably, the language is data-oriented, and not an imperative language such as Lims's.
- the present system allows for a later assigning or wild card connection.
- the "with" statement has found a very practical as users have been found to often describe circuits in terms of clusters of component connected to a central main component.
- the preprocessor provided allows for symbolic substitution, which can be parametrised in a later release, and to include text files in inline code. Especially, the "if" statement is introduced in this way. There is no need for an "if" statement to be built into the language.
- the library facility, by means of the referenced preprocessor, allows for a simple and flexible operation of the system; this advantage is enhanced in that the syntaxes of program and library are identical.
- the system can be provided with a trackwidth command. Also, comments may be fed through to the final output to act as instructions to the personnel executing the layout/assembly proper.
- a wire-wrap colour coding may be used.
- the trace facility has been found advantageous.
- the system provides for external signals, that are signals with no associated component.
- the combination of the two well-known programming languages has been found advantageous, although other languages could also be used. Specifically, the system set-up is such that no high-skill requirements are necessary for its operation.

Appendix 1

```
board Example euro_card
{
library 74TTL
{
elem selector = 74138;
elem inverter = 7404 ;
}
struct 2114
/*
1924 × 4 bit Static RAM
*/
{
shape = L3004 ;
1 = A6 ; 18 = +5v ;
2 = A5 ; 17 = A7 ;
3 = A4 ; 16 = A8 ;
4 = A3 ; 15 = A9 ;
5 = A0 ; 14 = D0 ;
6 = A1 ; 13 = D1 ;
7 = A2 ; 12 = D2 ;
8 = CS/ ; 11 = D3 ;
9 = GND ; 10 = WE/ ;
}RAM [1:2] ;
extern
AB[0:9], DB[0:7], sel/, WE/[1:2], GND, VCC ;
connections SIGNALS
{
trackwidth = default ;
```

Appendix 1-continued

```
with RAM [1:2]
{
~AB[0:9], A[0:9] ;
CS/ , 1/.selector ;
}
with RAM [1]
{
~DB[0:3], D[0:3] ;
~WE/[1], WE/ ;
}
with RAM [2]
{
~DB[4:7], D[0:3] ;
~WE/[2], WE/ ;
}
with selector
{
E1/, E2/, ~GND ;
A[0:2], ~AB[0:2] ;
}
~sel/, D1.inverter ;
Q1.inverter, E3.selector ;
}
connections POWER
{
trackwidth = default ;
~VCC, +5v* ;
}
connections GROUND
{
trackwidth = default ;
~GND, GND.* ;
}
}
```

Appendix 2

```
.PCB REDAC SHAPE FILE GENERATED BY CHOP V7.00
.REM INVOKED ON MON OCT 17 16:29:05 1983.
.TEXT
.CODE 1
EXAMPLE 17/10/83
.BOARD EURO_CARD
.COM
.REF
IC1 L3003
IC2 L3002
.REM
.REM 1024 × 4 BIT STATIC RAM
.REM
.COM
.REF
IC3 L3004
IC4 L3004
.EOD
.PCB REDAC CONNECTION FILE GENERATED
BY CHOP
.CON START OF SIGNALS CONNECTIONS
.CODE 2
IC3 5 IC4 5
IC3 6 IC4 6
IC3 7 IC4 7
IC3 4 IC4 4
IC3 3 IC4 3
IC3 2 IC4 2
IC3 1 IC4 1
IC3 17 IC4 17
IC3 16 IC4 16
IC3 15 IC4 15
IC3 8 IC4 8
IC3 8 IC1 14
IC1 4 IC1 5
IC2 2 IC1 6
.CON START OF POWER CONNECTIONS
.CODE 6
IC1 16 IC2 14
IC1 16 IC3 17
IC1 16 IC4 18
.CON START OF GROUND CONNECTIONS
.CODE 7
IC1 8 IC2 7
```

Appendix 2-continued

```
IC1 8 IC3 9
IC1 8 IC4 9
.REM CONNECTING EXTERNAL SIGNALS
IC1 4 IC1 8
IC1 1 IC4 5
IC1 2 IC3 6
IC1 3 IC3 7
.EOD
```

We claim:

1. An arrangement for generating a component interconnection list for an electrical circuit, each interconnection being in the form of a link between a first individual component pin and a second individual component pin, said arrangement comprising means for entering

a component list containing any component used,

a component description of any component used inclusive of a functional pin description thereof, and

a circuit interconnection description containing elements of the form of a first array of pins of a second array of named components connected to a third array of pins of a fourth array of named components, at least one array being non-unitary,

wherein for any circuit interconnection description element in which both said first and said third arrays are non-unitary the first and third arrays have equal numbers of array elements, said arrangement furthermore containing library means for storing said component descriptions as standard component descriptions, processing means for processing, under program control, the circuit interconnection description and the list of components used while addressing said library means for accessing said component descriptions, and output means for after said processing outputting said component interconnection list.

2. An arrangement as claimed in claim 1 wherein at least one circuit interconnection description element has at least one indirect assignment (with) description block containing a Pascal language type "with" statement with respect to a fifth component array for thereto assigning a description for thereupon processing this fifth array by the processing means on the basis that any pin without an associated specified component in the circuit interconnection description is contained within said fifth array.

3. An arrangement as claimed in claim 1 or 2 wherein said circuit interconnection description further contains at least one element in the form of a sixth array of equally named pins of any component to be connected to an associated common point in the electrical circuit.

4. An arrangement as claimed in claims 1 or 2, wherein said entering comprises a declaration of all components and named signals by listing them and assigning signal names to the pins thereof prior to activating said processing means for processing the circuit description.

5. An arrangement as claimed in claims 1 or 2, in which the processing means are controlled for executing a predetermined instruction to replace a first description by a second description wherever the first description occurs.

6. An arrangement as claimed in claims 1 or 2 in which the processing means are operative for under control of a second predetermined instruction inserting the contents of a specified file at a predetermined position within the program.

7. An arrangement as claimed in claims 1 or 2, wherein said output means are visualising output means.

8. A method of generating component interconnection lists for an electrical circuit in the form of an individual component pin to individual component pin instruction comprising forming a circuit interconnection description of the form

X[a:b].A[c:d], Y[e:f].B[g:h];

where

X and Y are component pin function descriptions,

A and B are component names,

a, b, c, d, e, f, g, and h are integers or letters which may take any value provided that the difference between a and b is equal to the difference between e and f unless either of those differences is equal to zero in which case the other may take any value,

and the comma represents "is connected to";

forming a list of components used and a description of those components; storing a library of standard component descriptions; and processing the circuit description and selected component descriptions from the library to produce an interconnection list in the form of an individual component pin to individual component pin connection.

9. A method according to claim 8, in which all components and signals are declared prior to processing the circuit description.

* * * * *

Teresa M. Corbin (SBN 132360)
Christopher Kelley (SBN 166608)
Thomas C. Mavrakakis (SBN 147674)
HOWREY SIMON ARNOLD & WHITE, LLP
301 Ravenswood Avenue
Menlo Park, California 94025
Telephone: (650) 463-8100
Facsimile: (650) 463-8400

Attorneys for Synopsys, Inc. and
Aeroflex, Inc., et al.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

RICOH COMPANY LTD., Plaintiff, v. AEROFLEX, INC., ET AL., Defendant.	Case No. C-03-4669 MJJ (EMC) Case No. C-03-2289 MJJ (EMC) **PROOF OF SERVICE OF ATTACHMENT 3 TO EXHIBIT A TO JOINT CLAIM CONSTRUCTION AND PREHEARING STATEMENT**
SYNOPSYS, INC., Plaintiff, v. RICOH COMPANY, LTD., Defendant.	

STATE OF CALIFORNIA)
) ss.:
COUNTY OF SAN MATEO)

I am employed in the County of San Mateo, State of California. I am over the age of 18 and not a party to the within action. My business address is 301 Ravenswood Avenue, Menlo Park, CA 94025.

On July 15, 2004, I served the within:

ATTACHMENT 3 TO EXHIBIT A TO JOINT CLAIM CONSTRUCTION AND PREHEARING STATEMENT

by placing true copies thereof in a sealed envelope(s) addressed as stated below and causing such envelope(s) to be delivered via:

☒ (PERSONAL SERVICE) I caused each such envelope to be delivered by hand to the offices of the interested party below.

Jeffrey B. Demain
Jonathan Weissglass
Altshuler, Berzon, Nussbaum, Rubin & Demain
177 Post Street, Suite 300
San Francisco, CA 94108

☒ (OVERNIGHT DELIVERY) by depositing in a box or other facility regularly maintained by Federal Express, an express service carrier, or delivering to a courier or driver authorized by said express service carrier to receive documents, a true copy of the foregoing document in sealed envelopes or packages designated by the express service carrier, addressed as stated below, with fees for overnight delivery paid or provided for and causing such envelope(s) to be delivered by said express service carrier.

Edward A. Meilman
Dickstein Shapiro Morin & Oshinsky, LLP
1177 Avenue of the Americas
New York, NY 10036

Gary M. Hoffman
Dickstein Shapiro Morin & Oshinsky, LLP
2101 L Street NW
Washington, DC 20037

I declare under penalty of perjury that I am employed in the office of a member of the bar of this Court at whose direction the service was made and that the foregoing is true and correct.

Executed on July 15, 2004, at Meno Park, California.

Susan J. Crane	/s/
(Type or print name)	(Signature)

Teresa M. Corbin (SBN 132360)
Christopher Kelley (SBN 166608)
Thomas C. Mavrakakis (SBN 147674)
Erik K. Moller (SBN 177927)
HOWREY SIMON ARNOLD & WHITE, LLP
301 Ravenswood Avenue
Menlo Park, CA 94025
Telephone: (650) 463-8100
Facsimile: (650) 463-8400

Attorneys for Defendants AEROFLEX INCORPORATED, et al. and SYNOPSYS, INC.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

RICOH COMPANY, LTD., Plaintiffs, v. AEROFLEX INCORPORATED, AEROFLEX COLORADO SPRINGS, AMI SEMICONDUCTOR, INC., MATROX ELECTRONIC SYSTESM, LTD., MATROX GRAPHICS, INC., MATROX INTERNATIONAL CORP., and MATROX TECH, INC., Defendants.	Case No. CV 03-04669 MJJ (EMC)
SYNOPSYS, INC., Plaintiff, vs. RICOH COMPANY, LTD., a Japanese corporation, Defendant.	Case No. CV-03-02289 MJJ (EMC) **DECLARATION AND SUMMARY OF OPINIONS OF DR. THADDEUS J. KOWALSKI ON CONSTRUCTION OF DISPUTED CLAIM TERMS OF UNITED STATES PATENT NO. 4,922,432**

DECLARATION AND SUMMARY OF OPINIONS OF DR. THADDEUS J. KOWALSKI ON CONSTRUCTION OF DISPUTED CLAIM TERMS OF UNITED STATES PATENT NO. 4,922,432

I. INTRODUCTION AND EXPERT QUALIFICATIONS

1. My name is Thaddeus J. Kowalski, Ph.D. I have prepared this report in connection with my role as an expert witness on behalf of Synopsys, Inc. and Aeroflex, Inc., Aeroflex Colorado Springs, Inc., AMI Semiconductor Inc., Matrox Electronic Systems, Ltd., Matrox Graphics Inc., Matrox International Corp., and Matrox Tech, Inc. in Case Nos. C-03-2289-MJJ and C-03-4669-MJJ (collectively "Synopsys and Defendants") regarding United States Patent No. 4,922,432 ("the '432 patent"). I have summarized in this section my educational background, career history, publications, and other pertinent qualifications. A copy of my current resume is also attached as Attachment 1 to Exhibit A to the Joint Claim Construction and Prehearing Statement.[1]

2. I am the owner of Software Practices and Technology, LLC, a premier provider of technology evaluations, merger and acquisition evaluations, business turnarounds, business and technology strategy, and software and process development. Previously, I was the Vice President and Chief Technology Officer for Advanced Real-Time Communications Architecture and Strategy at AT&T Labs. I was responsible for incubation of new business services. While at AT&T, I led R&D efforts in support of numerous business and consumer initiatives.

3. I hold a master's degree in computer engineering as well as a doctorate in electrical engineering from Carnegie Mellon University. I authored a book on design automation, authored a book on rule-based programming, filed two dozen significant patents, published almost a hundred journal articles, and have given over two hundred technical presentations. As a researcher in Bell Telephone Laboratories I made significant contributions in the areas of artificial intelligence, operating systems, computer-aided design, programming and text-processing environments, speech processing, and real-time systems. I am also a member of the "Vulcans" engineering honor-service society, Eta Kappa Nu, and Phi Beta Kappa.

---

[1] Referenced attachments are attached to Exhibit A of the Joint Claim Construction and Prehearing Statement.

4. For more than 15 years, from about 1980 to 1996, I was involved in the field of computer-aided design of integrated circuits. During those years I spent significant time in both the commercial research and academic environments. While I worked at Bell Telephone Laboratories I was also a Ph.D. student at Carnegie-Mellon University. Bell Telephone Laboratories is known through out the world for its groundbreaking work in computer-aided and VLSI design. Carnegie-Mellon was one of the two best Universities in the country in artificial intelligence and computer-aided design. My thesis work involved using rule-based expert system techniques to create a seminal expert system that took an algorithmic description using hardware description languages (HDLs) and created VLSI chips. After completing my thesis, I broadened and extended that work for nearly 10 years at Bell Telephone Laboratories. My work was known worldwide and called the Design Automation Assistant. This worldwide reputation in computer-aided design and verification led to the supervision of six Ph.D. students in four countries. In addition to my thesis and about a hundred journal articles, I published ``An Artificial Intelligence Approach to VLSI Design,'' which was reprinted four times. I also coauthored a book on advanced rule-based programming techniques entitled ``Rule Based Programming.''

5. Through this experience, as detailed above and as set forth in my resume I have acquired expertise in the areas of the creation of computer-aided design tools to assist in the design of integrated circuits and rule-based expert systems.

II. BACKGROUND TECHNOLOGY

6. I expect to provide background testimony regarding the state of the art of computer-aided design tools for designing integrated circuits using knowledge-based expert systems.

7. The processes of designing and manufacturing application specific integrate circuits (ASIC) are distinct processes that are both broad and complex. To make it easier to understand and give background, I will provide a description of the steps and processes to provide a helpful context. It is also my understanding from Synopsys' and Defendants' litigation counsel that Ricoh is claiming that the computer-aided design processes of the '432 patent are steps in manufacturing ASICs. In my expert opinion this is incorrect. Dr. Kobayashi, in a subsequent article, also has recognized that the

process of designing ASICs is separate from the processes for manufacturing an ASIC (CAD Tool Integration for ASIC Design: at 364-365 [Attachment 21]).

8. I will first explore the processes of designing ASICs and then briefly outline the processes of manufacturing ASICs. Designing ASICs can be described in two major steps, *i.e.*, specification capture followed by design synthesis and verification. Specification capture starts with a customer's need and a set of constraints. A designer will decide if the constraints will allow the customer's need to be realized as software on a general-purpose processor or as an application specific integrated circuit. If the constraints guide the designer to an ASIC design, a designer will typically codify the algorithm for the ASIC using some system to capture the operations and logical flows that show how to sequence the operations and specify them. The designer can also provide more detailed data that explicitly groups a sequence of operations to individual clock cycles. Next, the captured specification is synthesized and verified during design synthesis and verification. This step encompasses all the design phases, which successively map the captured specification into the photomask data. This step can be further subdivided into system, architecture, register transfer, logic, transistor, and layout levels. As the specification is mapped through these levels, technology, functional partitioning, module specification, and logic specification decisions are made. The logic is then placed and routed. The layout information is verified for consistency, correctness, and timing. The layout information is then rendered in a common physical design format so that the photomasks can be manufactured.

9. Before the manufacturing processes can begin a complex and costly process of creating photomasks is undertaken. This process transforms the physical design information into detailed instructions for the machine that creates the areas of light and dark on the photomasks. The photomasks are used over and over again in the manufacturing processes to create the transistor parts, circuit elements, insulating layers, and metallization paths necessary to fabricate the ASIC.

10. Manufacturing ASICs can be described in two major steps, *i.e.*, fabrication and the step of testing and verification. The fabrication process can be further subdivided into hundreds of phases. Highlights of these phases include: creating transistor parts, circuits elements, insulating layers, and

metallization paths by depositing layers of metal, oxide, and polysilicon on a substrate. Other highlights include thermal oxidation, lithography, etching, ion implantation, thermal redistribution, insulation, and metallization. The last step of the manufacturing processes tests and verifies the chip for environmental, stress, and defect problems.

III. INFORMATION CONSIDERED IN FORMING OPINIONS

11. In forming the opinions set forth in this report, I have considered and reviewed the text of the '432 patent (attached to the Joint Claim Construction Statement as Attachment 2); the history of the prosecution of the application that led to the issuance of the '432 patent (attached to the Joint Claim Construction Statement as Attachment 3); and the prior art references that were before the patent office during the prosecution of the '432 patent (a few of which are attached to the Joint Claim Construction Statement as Attachments 4, 6-15, and 35. I also considered the related '016 patent (Attachment 5) and '669 patent (Attachment 23), which also relate to rule-based expert systems and lists Dr. Kobayashi (an inventor on the '432 patent) as an inventor. I also considered the November 1989 article authored by Dr. Kobayashi (Attachment 22), which in my expert opinion, describes the same basic KBSC system described in the '432 patent. Finally, I have also considered the dictionaries, technical dictionaries, technical treatises and textbooks, and technical articles identified in Synopsys' and Defendants' portion of the Joint Claim Construction Chart. The relevant portions of these are provided in Attachments 16-21 and 24-34.

IV. OPINION AS TO LEVEL OF SKILL OF ONE OF ORDINARY SKILL IN THE ART

12. It is my understanding from Synopsys' and Defendants' litigation counsel that the proper construction of disputed claim terms requires determining the meaning of those disputed claim terms through the eyes of a person of ordinary skill in the art, *i.e.*, the pertinent scientific area, at the time the application for the '432 patent was filed, *i.e.*, January 1988. It is also my understanding from Synopsys' and Defendants' litigation counsel that the level of skill of a person of skill in the art may be determined based on one or more of the following factors: the educational level of the inventor; the type of problems encountered in the art; prior art solutions to those problems; the rapidity with which

innovations are made; sophistication of the technology; and the educational level of active workers in the field.

13. For purposes of interpreting the claims of the '432 patent, the pertinent scientific area is the creation of computer-aided design tools to assist in the design of integrated circuits. Typically, individuals of ordinary skill in this field would have a college degree (B.S. or M.S.) in a discipline such as Computer Science, Electrical Engineering, or Electrical and Computer Engineering and/or significant hands-on experience in creating large-scale software systems for assisting in the design of integrated circuits. In addition, individuals of ordinary skill in this field would have significant knowledge in programming, databases, user interfaces, and some knowledge of digital integrated circuit design.

V. OPINIONS AS TO MEANING OF DISPUTED CLAIM TERMS IN THE '432 PATENT

14. It is also my understanding from Synopsys' and Defendants' litigation counsel that the claim terms should be construed in light of the intrinsic evidence of record, including the patent specification, drawings in the patent, the prosecution (or file) history, and prior art cited in the patent or file history. I have reviewed and relied upon materials listed in Section III. Using these resources, my knowledge of the field into which the patented invention falls and my own familiarity with the literature on computer-aided design tools to assist in the design of integrated circuits both prior to and after the filing date of the patent, and my familiarity with the level of ordinary skill in the art at the time the patent was filed, I have formed an opinion as to how one of ordinary skill in the art would have interpreted the claim terms at the time the application for the '432 patent was filed. My opinion as to each of the terms identified to me as being at least potentially in dispute by the parties to this case are provided in the following paragraphs.

15. The proper construction and support as to each of the terms identified to me as being at least potentially in dispute by the parties to this case are provided in Synopsys' and Defendants' portion of the Joint Claim Construction Chart. I arrived at these constructions through discussions between Synopsys' and Defendants' litigation counsel and myself. These constructions are also provided in the following paragraphs with some additional comments or discussion, where appropriate.

A. **"A computer-aided design process for designing"**

16. The meaning of this phrase is "a process that uses a computer for designing, as distinguished from a computer-aided manufacturing process, which uses a computer to direct and control the manufacturing process." This definition is not only supported by the definitions for computer-aided design (CAD) and computer-aided manufacturing (CAM) provided by Synopsys and Defendants, but is also supported by the dictionary definitions provided for those terms by Ricoh (IBM Dictionary at 129-130 [Attachment 16]; IEEE Dictionary at 180 [Attachment 19]). It is also consistent with the ordinary dictionary definitions for "designing" and "manufacturing" (Webster's Ninth New Collegiate Dictionary at 343, 725 [Attachment 20]).

17. Neither the '432 patent's specification nor its file history alters the ordinary meaning of this phrase. In fact, the '432 patent specification is consistent with and supports the ordinary meaning as set forth in the previous paragraph ('432 patent: 1:9-12 [Attachment 2]).

18. I disagree with the definition proposed by Ricoh for this phrase: "During the manufacture of a desired application specific integrated circuit (ASIC) chip that is designed to perform a specific purpose, a process of designing the desired ASIC using a computer." In particular, I disagree with Ricoh's attempt to equate or include computer-aided design processes for designing ASIC's with manufacturing ASICs. Designing an ASIC is a predicate process that creates the plan to make an ASIC. Specifically, the claimed processes of the '432 patent only produces the list of the necessary parts and their required interconnections and then uses that list to pictorially represent the physical placement of these necessary parts and the routing of their required interconnections. After additional complex processes, this information is eventually used to make the photomasks, or masks, that are used in the processes that manufacture the ASIC. The goal of the design process is to only have to make these masks once because of their cost, which for present day ASICs is about one million dollars. Manufacturing an ASIC entails the processes that physically implement the plan each time the ASIC is manufactured. Unlike the design processes, which are performed once, the processes that manufacture ASICs are repeated each time an ASIC is manufactured.

B. **"application specific integrated circuit"**

19. The meaning of this phrase is "an interconnected miniaturized electronic circuit on a single piece of semiconductor material designed to perform a specific function, as distinguished from standard, general purpose integrated circuits, such as microprocessors, memory chips, etc." The '432 patent specifically defines this phrase as "an integrated circuit chip designed to perform a specific function, as distinguished from standard, general purpose integrated circuits, such as microprocessors, memory chips, etc." ('432 patent: 1:13-17 [Attachment 2]). Synopsys' and Defendants' definition for this phrase is the same but merely expands on this definition by providing a definition for "integrated circuit."

20. I disagree with Ricoh's definition, "an integrated circuit chip designed to perform a specific function," because it fails to explicitly exclude "general purpose integrated circuits, such as microprocessors, memory chips, etc." Ricoh's definition is contrary to the '432 patent, which provides the ordinary meaning for this phrase ('432 patent: 1:13-17 [Attachment 2]).

C. **"actions and conditions"**

21. "Actions and conditions" refer to "logical operations" or more specifically "logical steps and decisions." This is consistent with the definitions for "operations" provided by Synopsys and Defendants in their portion of the Joint Claim Construction Chart (IBM dictionary: at 479 [operations] [Attachment 16]). It is also consistent with the '432 patent's specification ('432 patent: 2:24-27; 4:15-19; 6:3-14; 8:47-51 [Attachment 2]). The limitation that these logical steps and decisions be "represented as rectangles and diamonds in the flowchart" is dictated by the '432 patent's file history. In particular, the '432 patent's file history unambiguously limits the input format to a flowchart format ('432 patent's file history: April 1989 Amendment at 9, 11; October 1989 Examiner Interview Summary; November 1989 Amendment at 6-7 [Attachment 3]). This is also consistent with the fact that the only adequately described input format in the '432 patent is the flowchart format ('432 patent: 3:50-59; 4:61-63; 7:20-23 [Attachment 2]). Although the '432 patent mentions a list type format ('432 patent: 2:21-24 [Attachment 2]) and appears to show a statelist in Appendix A ('432 patent: 14:7-30 [Attachment 2]), there is no description explaining that format in the '432 patent. In fact, such

explanation for the list format was only added in the later-filed, and related, '016 patent ('016 patent: 7:32-9:52 [Attachment 5]).

D. **"architecture independent"**

22. The phrase "architecture independent" was not in the originally filed application for the '432 patent. It was added to the application for the '432 patent in the abstract, the specification, and the claims as part of the April 1989 Amendment in the '432 patent's file history (April 1989 Amendment at pages 1-8 [Attachment 3]). This phrase is vague and imprecise and does not have a clear meaning. In addition, the '432 patent also does not provide a definition for this phrase.

23. "Architecture independent" is narrowly defined in the '432 patent's file history by the patent applicant's referencing and incorporating of the prior art '435 patent (Darringer et al.). Based on the file history and the '435 patent this phrase means: "not including (*i.e.*, excluding) a register transfer level (RTL) description or any other description that is hardware architecture dependent. An RTL description consists of: 1) defining the inputs, outputs, and any registers of the proposed ASIC; and, 2) describing for a single clock cycle of the ASIC how the ASIC outputs and any registers are set according to the values of the ASIC inputs and the previous values of the registers; an RTL description defines any control needed for the ASIC" ('432 patent's file history: November 1989 Amendment at 7 [Attachment 3]). Specifically, the prior art '435 patent does describe an input specification that includes logical operations, *i.e.*, actions and conditions ('435 patent: Fig. 4; 4:26-32; 5:27-35 [Attachment 4]). The '432 patent claims were claimed by the patent applicant to be different from the '435 patent because, among other things, the phrase "architecture independent" was added to the claims. The patent applicant also stated that the prior art '435 patent's input specification included a register-transfer level description and therefore, that the '435 input specification was hardware architecture dependent, in contrast to the patent applicant's architecture independent specification (April 1989 Amendment at 8-10, 13; November 1989 Amendment at 7 [Attachment 3]).

24. In other words, the input specification of the prior art '435 patent was claimed to be different from the input specification of the '432 patent because, unlike the prior art '435 patent, the

input specification of the '432 patent only consists of logical operations and excludes a register-transfer level description.

25. I disagree with Ricoh's proposed definition for "architecture independent actions and conditions" and Ricoh's incorporation of that definition into the definition of the phrase "a set of definitions of architecture independent actions and conditions." Ricoh's definition provides: "functional or behavioral aspects of a portion of a circuit (or circuit segment) that does not imply any set architecture, structure, or implementing technology." First, Ricoh's proposed definition fails to acknowledge the limitations from patent applicants statements in the '432 patent's file history. As discussed in Paragraphs 21 and 23, these statements limit the input specification and therefore, the "actions and conditions" to a flowchart format. Second, Ricoh's proposed definition fails to recognize that "architecture independent" was narrowly defined in the '432 patent's file history as excluding a register-transfer level description as defined in the prior art '435 patent. Without the file history statement, the phrase "architecture independent" is vague and imprecise and is not defined in the '432 patent. Third, Ricoh appears to equate "architecture independent" with the equally imprecise and vague phrase "does not imply any set architecture, structure, or implementing technology" and which is also contrary to the definition claimed in the '432 patent's file history and not supported by any description in the '432 patent's specification. Finally, Ricoh's definition uses the vague and imprecise phrase "functional or behavioral aspects of a portion of a circuit," which is neither consistent with nor supported by the description in the '432 patent (2:24-27; 4:15-19; 6:3-14; 8:47-51 [Attachment 2]).

26. I also do not agree with Ricoh's definition for the word "storing" in a number of the claim steps: "placing in computer memory." In particular, "storing" means "placing data in any storage device" that is accessible by the processor in the computer system (IBM Dictionary of Computing at 654 [Attachment 16]).

E. **"a set of definitions of architecture independent actions and conditions"**

27. This entire phrase, incorporating the above definitions, means: "a set of named descriptions defining the functionality and arguments for the available logical steps and decisions that may be specified in the flowchart; and excluding a register transfer level description." This is

consistent with the '432 patent's specification ('432 patent: 4:61-63; 5:20-22; 6:3-14; 7:25-50; 8:47-51[Attachment 2]).

28. I disagree with Ricoh's definition for this entire claim step. That disagreement stems from Ricoh's incorporations of its definitions for "storing" and "architecture independent actions and conditions" as discussed in Paragraphs 25-26.

F. **"hardware cells"**

29. "Hardware cells" refers to the "logic blocks for which the functional level (e.g., register transfer level), logic level (e.g., flip flop and gate level), circuit level (e.g., transistor level), and layout level (e.g., geometrical mask level) descriptions are all defined." This definition is consistent with the '432 patent's requirement that "previously designed, tested, and proven hardware cells" are defined in the cell library ('432 patent: 5:15-20 [Attachment 2]). The '432 specification and the other steps in claims 13-17 also require that the hardware cells must be defined with the following types of information: functional level, logic level, circuit level, and layout level ('432 patent: 2:34-39; 3:59-67; 9:24-51; 16:60-68 [Attachment 2]).

30. My disagreement with Ricoh's definition for this term, "previously designed circuit components or structure that have specific physical and functional characteristics used as building blocks for implementing an ASIC to be manufactured" includes the use of the phrase "specific physical and functional characteristics." The '432 patent's described method for designing an application integrated circuit requires the hardware cells in the cell library to be previously designed, tested, and proven logic blocks ('432 patent: 5:15-20 [Attachment 2]). This means that those hardware cells must be defined with the following types of information: functional level, logic level, circuit level, and layout level ('432 patent: 2:34-39; 3:59-67; 9:24-51; 16:60-68 [Attachment 2]). I also disagree with the use of the phrase "for implementing an ASIC to be manufactured." As explained in Paragraphs 16-18, the '432 patent claims 13-17 encompass computer-aided design processes for designing an ASIC and not manufacturing them.

G. **"data describing a set of available integrated circuit hardware cells for performing the actions and conditions defined in the stored set"**

31. This claim phrase means "a set of named integrated circuit hardware cells that includes at least one hardware cell for each stored definition that may be specified for the available logical steps and decisions; where each named hardware cell has corresponding descriptions at the functional level (e.g., register transfer level), logic level (e.g., flip-flop and gate level), circuit level (e.g., transistor level), and layout level (e.g., geometrical mask level) that are all defined." The '432 patent specification requires that there be one or more defined hardware cells corresponding to each of the stored definitions that may be specified ('432 patent: Fig. 4; 5:23-25 [Attachment 2]). The '432 specification and the other steps in claims 13-17 also require that the data describing the hardware cells must define the following types of information: functional level, logic level, circuit level, and layout level ('432 patent: 2:34-39; 3:59-67; 9:24-51; 16:60-68 [Attachment 2]).

32. I disagree with Ricoh's proposed definition, "a library of cell information that describe hardware cells capable of performing the different architecture independent actions and conditions placed in the library of definitions," because Ricoh fails to include a requirement that there is at least one corresponding hardware cell description for each of the stored definitions that may be specified. Similar to its definition for hardware cell, Ricoh's definition again fails to specify what constitutes the "data describing" the available hardware cells as I explained in Paragraph 33.

H. **"expert system"**

33. The meaning of an expert system is: "software executing on a computer system that attempts to embody the knowledge of a human expert in a particular field and then uses that knowledge to simulate the reasoning of such an expert to solve problems in that field. This system is comprised of a knowledge base containing rules, working memory containing the problem description, and an inference engine. It solves problems through the selective application of the rules in the knowledge base to the problem description, as distinguished from conventional software, which uses a predefined step-by-step procedure (algorithm) to solve problems." This ordinary meaning is apparent from the texts, dictionaries, and articles from that time period as well as the Dr. Kobayashi's own

article from the same time his application for the '432 patent was pending (Computer Aided VLSI Design Vol.1 No. 4: 351, 377-381, 383, 388-389 [Attachment 22]; '669 patent: Abstract, 1:13-16, 2:28-33, 4:31-62; 5:21-38, 5:58-68, 6:1-7:68, 17:62 [Attachment 23]; Understanding Expert Systems: 7-10, 29-30, 40, 42, 74-78, 99-110 [Attachment 24]; An Artificial Intelligence Approach To VLSI Design: at 9-15 [Attachment 25]; Artificial Intelligence Terminology: at 6 [algorithm], 10 [antecedent], 53 [consequent], 86-87 [expert system], 127 [inference engine], 140 [knowledge based system], 204 [production system], 223 [rule base, rule-based system], 281 [working memory] [Attachment 26]; Microsoft Press Computer Dictionary: at 136 [expert system] [Attachment 27]; IBM Dictionary of Computing: 591 [rule interpreter, rule-based system] [Attachment 16]; The VLSI Design Automation Assistant: at 34 [Attachment 28]; A Rule-Based Logic Circuit Synthesis System for CMOS Gate Arrays: at 597 [Attachment 29]; Expert Systems: A Non-Programmer's Guide: 8-10, 16 [Attachment 30]; Expert Systems: Tools and Applications: at 269 [Attachment 31]; Expert Systems: Principles and Case Studies: at 11-12 [Attachment 32]; Knowledge-Based Systems: The View in 1986: at 16 [Attachment 33]). In particular, Mr. Kobayashi's article states that there are only "two different types of approaches to automatic logic synthesis: algorithmic and rule-based. IF-THEN-type rules rather than algorithmic programming languages are used in the latter approach to synthesize logic circuits" (Computer Aided VLSI Design Vol.1 No. 4: at 351 [Attachment 22]). Mr. Kobayashi also states that the KBSC method described in this article and in his patent "is clearly distinguished from other logic synthesis systems in terms of its flowchart input form and rule-based approach to automatic data-path and control logic synthesis" (Computer Aided VLSI Design Vol.1 No. 4: at 389 [Attachment 22]).

34. The '432 patent's specification and the claims also support this ordinary meaning for the term expert system ('432 patent: 2:58-63; 5:6-8; 8:58-60; 10:39-11:26; 14:50-59; 15:53-58[Attachment 2]).

35. The '432 patent's file history and the prior art references referred to in that file history also dictate this definition for the term "expert system" (April 1989 Amendment at 9-11, 15, 17; October 1989 Examiner Interview Summary; November 1989 Amendment at 7, 9 [Attachment 3]; '435 patent: 7:32-9:35 [Attachment 4]; An Overview of Logic Synthesis Systems: at 170 [Attachment

7]; The CMU Design Automation System: at 75-77 [Attachment 8]). In particular, the patent applicant states numerous times and the patent examiner agreed in the '432 patent's file history that a rule-based expert system was required for translating the flowchart into a netlist. I would also like to point out that in the 1988 time frame the terms "expert system," "rule-based expert system," "knowledge based expert system," and "production system" were all used interchangeably to refer to an "expert system" as defined in Paragraph 33.

I. **"knowledge base"**

36. The meaning of the term "knowledge base" is "the portion of the expert system containing a set of rules embodying the expert knowledge for the particular field." This ordinary meaning is apparent from the texts from the 1988 time period as well as Dr. Kobayashi's own related '669 patent, which was filed about the same time as his application for '432 patent was filed ('669 patent: Abstract [Attachment 23]).

37. The '432 patent's specification and the claims also support this ordinary meaning for the term knowledge base ('432 patent: 10:39-11:15 [Attachment 2]).

38. The '432 patent's file history and the prior art references referred to in that file history also dictate this definition for the term "knowledge base" ('432 patent's file history: April 1989 Amendment at 9-11, 15, 17; October 1989 Examiner Interview Summary; November 1989 Amendment at 7, 9 [Attachment 3]; '435 patent: 7:32-9:35 [Attachment 4]).

39. I disagree with Ricoh's definition for the term "expert system knowledge base," which provides: "database used to store expert knowledge of highly skilled VLSI designers." First, Ricoh's definition fails to acknowledge that a knowledge base is just one portion of the rule-based expert system; specifically, the portion containing the IF-THEN rules embodying the expert knowledge ('432 patent: 14:50-59; 15:53-58 [Attachment 2]). The file history requires that a rule-based expert system be used (April 1989 Amendment at 9-11, 15, 17; October 1989 Examiner Interview Summary; November 1989 Amendment at 7, 9 [Attachment 3]). Second, Ricoh's definition fails to include a definition for "expert system" and all of the necessary portions required for such a system based upon the ordinary meaning of "expert system" set forth in Paragraph 33. Third, Ricoh's definition is

inaccurate because it fails to require that the knowledge base store the expert knowledge in the form of IF-THEN rules. This is contrary to the '432 patent specification, the other claims, the '432 patent's file history, and the inventors own article and his related patent from that period (Computer Aided VLSI Design Vol.1 No. 4: 351, 377-381, 383, 388-389 [Attachment 22]; '669 patent: Abstract, 1:13-16, 2:28-33, 4:31-62; 5:21-38, 5:58-68, 6:1-7:68, 17:62 [Attachment 23]).

40. Ricoh's definition also contradicts the patent applicant's own claims that the prior art '435 patent (Darringer et al.) differed because it did not provide a "knowledge base of any kind" or a "rule-based expert system" ('432 patent's file history: April 1989 Amendment at 9-10 [Attachment 3]). The prior art '435 patent contained "expert knowledge of highly skilled VLSI designers," which is referred to as transforms ('435 patent: 7:32-9:35 [Attachment 4]). Ricoh's definition is too broad because it would encompass the "transforms" described in the prior art '435 patent, which the patent applicant claimed were not a knowledge base.

J. **"a set of rules for selecting hardware cells to perform the actions and conditions"**

41. The meaning of this phrase is: "a set of rules, each having an antecedent portion (IF) and a consequent portion (THEN), embodying the knowledge of expert designers for application specific integrated circuits, which enables the expert system to map the specified stored definitions for each logical step and decision represented in the flowchart to a corresponding stored hardware cell description." This definition is consistent with the ordinary meaning of the term "rules" when referring to the rules that are contained in the knowledge base of a rule-based expert system (Expert Systems: A Non-Programmer's Guide: 8-10, 16 [Attachment 30]; Expert Systems: Principles and Case Studies: at 11-12 [Attachment 32]). The definition is also consistent with the inventor's article published while the '432 patent application was pending (Computer Aided VLSI Design Vol.1 No. 4: 351, 377-381, 383, 388-389 [Attachment 22]). In particular, the quote from Paragraph 33 of the Dr. Kobayashi's article shows that the KBSC method described there and in the '432 patent was limited to using IF-THEN type rule to codify the expert knowledge of ASIC designers.

42. The definition of this phrase is also consistent with the '432 patent's specification and required by the other claimed steps and other claims ('432 patent: 2:58-63; 8:20-30; 8:58-9:62; 10:39-11:26; 14:50-59; 15:53-58; 16:34-65 [Attachment 2]).

43. The '432 patent's file history and the prior art references, in that file history, also dictate the definition for this phrase ('432 patent's file history: April 1989 Amendment at 9-11, 15, 17; October 1989 Examiner Interview Summary; November 1989 Amendment at 7, 9 [Attachment 3]; '435 patent: 7:32-9:35 [Attachment 4]; An Overview of Logic Synthesis Systems: at 170 [Attachment 7]; The CMU Design Automation System: at 75-77 [Attachment 8]).

44. I disagree with Ricoh's definition for this phrase: "a plurality of rules for selecting among the hardware cells placed in the hardware cell library, wherein the rules comprise the expert knowledge of highly skilled VLSI designers formulated as prescribed procedures." First, Ricoh's definition of "rule" is contrary to the ordinary meaning for that term when applied to rule-based expert systems (Expert Systems: A Non-Programmer's Guide: 8-10, 16 [Attachment 30]; Expert Systems: Principles and Case Studies: at 11-12 [Attachment 32]). Second, Ricoh's definition for "rule" is also inconsistent with the '432 patent's file history. Specifically, Ricoh's definition for "rule" provides: "the expert knowledge of highly skilled VLSI designers formulated as prescribed procedures." This contradicts patent applicant's claims that the prior art '435 patent (Darringer et al.) differed because it did not provide a "knowledge base of any kind" or a "rule-based expert system" ('432 patent's file history: April 1989 Amendment at 9-10 [Attachment 3]). The prior art '435 patent does contain "expert knowledge of highly skilled VLSI designers formulated as prescribed procedures" ('435 patent: 7:32-9:35 [Attachment 4]). The '435 patent refers to the expert knowledge of highly skilled VLSI designers as "transforms" and the prescribed procedures as "scenarios." Ricoh's definition is too broad because it would encompass the "transforms" and "scenarios" described in the prior art '435 patent, which the patent applicant claimed were not a knowledge base.

45. I also disagree with Ricoh's definition for this phrase because Ricoh does not require that the set of "rules" be for mapping the specified stored definitions for each logical step and decision represented in the flowchart to a corresponding stored hardware cell description. This is inconsistent

with the '432 patent specification, claim 13, and the other claims ('432 patent: 2:58-63; 8:20-30; 8:58-9:62; 10:39-11:26; 14:50-59; 15:53-58; 16:34-65 [Attachment 2]).

K. **"describing for a proposed application specific integrated circuit a series of architecture independent actions and conditions"**

46. The meaning of this phrase is: "the designer represents a sequence of logical steps (rectangles) and decisions (diamonds), and the transitions (lines with arrows) between them in a flowchart format for a proposed application specific integrated circuit." This definition is consistent with the '432 patent's only adequately described input specification. This definition is dictated by the claim language in claim 13 and the patent applicant's statements in the '432 patent's file history ('432 patent: Figs. 1a, 5, &7; 2:21-27; 3:20-22; 3:50-59; 4:5-22; 4:35-38; 7:12-23; 16:34-65 [Attachment 2]; '432 patent's file history: April 1989 Amendment at 9, 11; October 1989 Examiner Interview Summary; November 1989 Amendment at 6-7 [Attachment 3]).

47. This definition is also consistent with the dictionary definitions of the words "series," "sequence," and "describing" (Webster's Ninth New Collegiate Dictionary at 1074, 1073, 343 [Attachment 20]).

48. I disagree with Ricoh's definition for this claim step, which provides: "a user describing an input specification containing the desired functions to be performed by the designed ASIC." First, Ricoh's definition incorrectly provides that the "desired functions to be performed" are contained in this step. This is contrary to the '432 patent's specification and the claim language. Second, Ricoh's definition ignores the words "series" and "conditions" and fails to acknowledge that this step is for describing, or representing, the sequence of logical steps and decisions and the transitions between them. This definition is also contrary to the '432 patent's specification and the claim language. Third, Ricoh's definition does not provide for the limitation required by the '432 patent's file history that the sequence of logical steps and decisions and the transitions between them must be represented or described in a flowchart format ('432 patent's file history: April 1989 Amendment at 9, 11; October 1989 Examiner Interview Summary; November 1989 Amendment at 6-7 [Attachment 3]).

49. Ricoh's definition is also incorrect because it leaves out critical information. Specifically, describing the sequence of the steps and decisions is necessary for producing the control signals required for the ASIC. This sequence information for the steps and decisions is critical because these steps and decisions are required to be "architecture independent," excluding a register-transfer level description.

L. **"specifying for each described action and condition of the series one of said stored definitions"**

50. The meaning of this phrase is: "the designer assigns one definition from the set of stored definitions for each of the described logical steps and decisions represented in the flowchart." This definition is consistent with the '432 patent's only adequately described input format and dictated by the claim language in claim 13 ('432 patent: Fig. 5; 3:20-22; 4:61-63; 5:20-22; 7:24-25; 8:23-26; 8:51-56 [Attachment 2]).

51. I disagree with Ricoh's definition for this claim step: "specifying for each desired function to be performed by the desired ASIC one of the definitions of the architecture independent actions and conditions stored in the library of definitions that is associated with the desired function." First, Ricoh's definition reiterates the same inaccuracies from the previous describing step. Second, Ricoh's definition contradicts the '432 patent's specification by failing to acknowledge that the designer separately assigns the stored definitions to each logical step and decision. In addition, the claim language clearly provides that this assigning step must occur after the series of logical steps and decisions have been described (or represented).

52. I also disagree with Ricoh's definition for "specifying": "mapping or associating a desired function to be performed by the manufactured ASIC with a definition from the library of definitions." The claim language and the '432 patent specification make clear that this step is performed by the user assigning the stored definitions to each of the described logical steps and decisions ('432 patent: Fig. 5; 3:20-22; 4:61-63; 5:20-22; 7:24-25; 8:23-26; 8:51-56 [Attachment 2]). This "specifying" step is part of input specification provided by the user for the desired ASIC to be designed and cannot be performed by the system described in the '432 patent. Ricoh's definition is

also contrary to the dictionary definitions of "specify" and "specification" Webster's Ninth New Collegiate Dictionary at 1132 [Attachment 20]). Finally, I also disagree with Ricoh's adding of the phrase "performed by a manufactured ASIC." As explained in Paragraphs 16-18, the '432 patent claims 13-17 encompass computer-aided design processes for designing an ASIC and not manufacturing them.

M. **"which corresponds to the desired action or condition to be performed"**

53. The meaning of this phrase is: "each specified definition must correspond to the intended step or decision to be performed." This definition is consistent with the ordinary meaning of the terms, the '432 patent specification, and required by the language of the claims ('432 patent: Fig. 5; 3:20-22; 4:61-63; 5:20-22; 7:24-25; 8:23-26; 8:51-56; 16:34-65 [Attachment 2]).

54. Ricoh's definition, which is provided in Paragraph 51, is incorrect because it replaces "correspond to" with the phrase "associated with." The claim language and the '432 patent specification and the ordinary meaning of these terms all require that the definitions assigned must correspond to (or match) what is intended for each logical step or decision (Webster's Ninth New Collegiate Dictionary at 110 [associate], 293 [correspond] [Attachment 20]).

N. **"selecting from said stored data for each of the specified definitions a corresponding integrated circuit hardware cell for performing the desired function of the application specific integrated circuit"**

55. The meaning of this phrase is: "mapping the specified stored definitions for each logical step and decision represented in the flowchart to a corresponding stored hardware cell description." This definition is consistent with the '432 patent's only adequately described system and dictated by the claim language in claim 13 ('432 patent: Fig. 4; 3:16-19; 4:66-5:3; 5:22-29; 8:31-37; 8:58-60; 9:52-60; 16:34-65 [Attachment 2]) '432 patent file history: April 1989 Amendment at 10 [Attachment 3].

O. **"said step of selecting a hardware cell comprising applying to the specified definition of the action or condition to be performed, a set of cell selection rules stored in said expert system knowledge base"**

56. The meaning of this phrase is: "the mapping of the specified definitions to the stored hardware cell descriptions must be performed by an expert system having an inference engine for selectively applying a set of rules, each rule having an antecedent portion (IF) and a consequent portion (THEN), embodying the knowledge of expert designers for application specific integrated circuits, which enables the expert system to map the specified stored definitions for each logical step and decision represented in the flowchart to a corresponding stored hardware cell description." This definition is consistent with the '432 patent's only adequately described system and dictated by the claim language in claim 13 and the patent applicant's statements in the '432 patent's file history ('432 patent: Abstract; 2:58-63; 5:6-8; 8:29-37; 8:58-60; 9:8-13; 11:16-26; 16:34-65 [Attachment 2]; '432 patent's file history: April 1989 Amendment at 8-11, 17; October 1989 Examiner Interview Summary; November 1989 Amendment at 4, 6-7, 9 [Attachment 3]; '435 patent: 7:32-9:35 [Attachment 4]).

57. Ricoh's combined definition for both N and O provides: "selecting from the plurality of hardware cells in the hardware cell library a hardware cell for performing the desired function of the desired ASIC through the application of the rules." This definition is not correct and leaves out numerous requirements. First, Ricoh's definition fails to require that the mapping be performed for "each specified definition." This is required by both the claim language itself and the '432 patent specification. Second, Ricoh's definition also fails to limit how the "mapping" is performed. Specifically, that the mapping must be accomplished using a rule-based expert system ("432 patent's file history: April Amendment at 10 [Attachment 3]). Third, Ricoh's definition is contrary to the ordinary meaning of "expert system" and the statements in the '432 patent's file history because it fails to require that the applying of the "set of cell selection rules" in the knowledge base of the rule-based expert system is performed by that rule-based expert system's inference engine (Paragraphs 33-35).

P. **"netlist"**

58. The meaning of this phrase is: "a structural description that includes a custom controller type hardware cell and all other hardware cells required to implement the application specific integrated circuit's operations and any necessary interconnections including the necessary control and data path information for connecting the hardware cells and the controller." This definition is consistent with the '432 patent's only adequately described system as well as the claims ('432 patent: Abstract; 1:17-37; 2:39-44; 4:39-43; 5:8-12; 5:30-40; 9:62-10:9; 12:31-35; 13:55-14:3; 16:34-65 [Attachment 2]).

59. Ricoh's definition, which is: "a description of the hardware components (and their interconnections) needed to manufacture the ASIC as used by subsequent processes, e.g., mask development, foundry, etc." is not correct. First, Ricoh's definition fails to require that there is a description for the controller type hardware cell, which is needed to provide the control for the other needed hardware cells as required by the '432 patent specification. Second, the '432 patent also clearly requires that the netlist must provide the structural description of the control paths and data paths that connect the controller type hardware cell and the other needed hardware cells.

60. I also disagree with the use of the phrase "needed to manufacture the ASIC as used by subsequent processes, e.g., mask development, foundry, etc." As explained in Paragraphs 16-18, the '432 patent claims 13-17 encompass computer-aided design processes for designing an ASIC and not manufacturing them.

Q. **"generating for the selected integrated circuit hardware cells, a netlist defining the hardware cells which are needed to perform the desired function of the integrated circuit"**

61. The meaning of this phrase is: "producing a list of the needed hardware cells by eliminating any mapped hardware cells that are redundant or otherwise unnecessary and producing a custom controller type hardware cell for providing the needed control for those other hardware cells." This definition is consistent with the '432 patent's only adequately described system and dictated by

the claim language in claim 13 ('432 patent: Abstract; 1:17-37; 2:39-44; 4:39-43; 5:8-12; 5:30-40; 9:62-10:9; 13:55-14:3; 16:34-65 [Attachment 2]).

62. Ricoh's definition, which is: "generating a netlist that identifies the hardware cells needed to perform the function of the desired ASIC and the necessary parameters for connecting the respective inputs and outputs of each hardware cell, the netlist is passed to the next subsequent step in the process for manufacturing the desired ASIC" is not correct. First, contrary to the claim language, "for the selected cells," Ricoh's definition fails to require that generation step must follow the prior step of selecting (mapping) the hardware cell descriptions and therefore, performed using those selected (mapped) hardware cell descriptions. Instead, Ricoh's definition, contrary to the claim language and the '432 patent's specification, provides that the generation step is performed without any connection to this prior step. Second, Ricoh's definition does not require that the controller type hardware cell be generated. This is also contrary to the requirements in the '432 patent's specification and the definition of "netlist" as required by the '432 patent's specification. Third, again contrary to the '432 patent's specification and the claim language, Ricoh's definition fails to require that the "hardware cells which are needed" are the selected (mapped) hardware cell descriptions that remain after any unnecessary or redundant hardware cell descriptions have been eliminated.

R. **"generating...interconnection requirements therefor"**

63. The meaning of this phrase is: "producing the necessary structural control paths and data paths for the needed hardware cells and the custom controller." This definition is consistent with and required by the claim language and the '432 patent's specification ('432 patent: Abstract; Figs. 6 & 13-15; 1:17-37; 2:39-44; 3:23-25; 3:40-45; 4:39-43; 5:8-12; 5:30-40; 9:62-10:9; 13:55-14:3; 16:34-65 [Attachment 2]).

64. I do not agree with the Ricoh's definition and specifically the part that provides: "the necessary parameters for connecting the respective inputs and outputs of each hardware cell." The "interconnection requirements" for the needed hardware cells netlist must include the structural control paths and data paths connecting those cells. Otherwise, it would not be a netlist as defined in Paragraph 58.

S. **"generating from the netlist the mask data required to produce an integrated circuit having the desired function"**

65. The meaning of this phrase is: "producing, from the structural netlist, the detailed layout level geometrical information required for manufacturing the set of photomasks that are used by the processes that directly manufacture the application specific integrated circuit." This definition is consistent with the ordinary meaning of the phrase "mask data" and is also consistent with the minimal information provided by the '432 patent specification on what is meant by "mask data" ('432 patent: Abstract; Fig. 1c; 1:42-44; 1:54-58; 2:44-49; 4:44-46; 5:40-46; 14:4-6; 16:34-68 [Attachment 2]).

66. Ricoh's definition, which is: "producing from the netlist of hardware cells to be included in the designed ASIC mask data which can be directly used by a chip foundry in the fabrication of the ASIC" is not correct. First, Ricoh's definition does not define "mask data." Second, "mask data" is only the information or data that can be used by other complex processes for making photomasks for the designed application specific integrated circuit. Mask data cannot be used to fabricate or manufacture an application specific integrated circuit.

T. **"generating data paths for the selected integrated circuit hardware cells"**

67. The meaning of this phrase is: "producing the necessary structural descriptions of the data paths for the selected hardware cells." This definition is consistent with the ordinary meaning and is also consistent with the '432 patent specification ("432 patent: Abstract; 2:39-40; 4:63-66; 5:6-12; 5:30-37; 6:29-31; 6:37-43; 6:50-53; 9:62-10:9; 13:55-14:3; 16:34-17:3 [Attachment 2]).

68. Ricoh's definition, which is: "producing signal lines for carrying data to the hardware cells" is not correct. First, Ricoh's definition ignores the claim language "for the selected integrated circuit hardware cells." Second, the processes of the '432 patent are for producing design data and not the physical "signal lines for carrying data" (IEEE Standard Dictionary at 898 [signal line] [Attachment 19]). As explained in Paragraphs 16-18, the '432 patent claims 13-17 encompass computer-aided design processes for designing an ASIC and not manufacturing them.

U. **"said step of generating data paths comprises applying to the selected cells a set of data path rules stored in a knowledge base and generating the data paths therefrom"**

69. The meaning of this phrase is: "the generating step must be performed by at least an expert system having an inference engine for selectively applying a set of rules, each having an antecedent portion (IF) and a consequent portion (THEN), embodying the knowledge of expert designers for application specific integrated circuits, which enables the expert system to produce the necessary data paths for the mapped hardware cells." This definition is consistent with the '432 patent's only adequately described system and dictated by the claim language in this claim and the patent applicant's statements in the '432 patent's file history ('432 patent: Abstract; 5:6-12; 9:62-10:9; 13:55-14:3 [Attachment 2]; '432 patent's file history: April 1989 Amendment at 9-11, 15, 17; October 1989 Examiner Interview Summary; November 1989 Amendment at 7, 9 [Attachment 3]; '435 patent: 7:32-9:35 [Attachment 4]; An Overview of Logic Synthesis Systems: at 170 [Attachment 7]; The CMU Design Automation System: at 75-77 [Attachment 8]).

70. Ricoh's definition provides: "wherein the step of producing signal lines for carrying data comprises applying rules, which are placed in computer memory, to produce the signal lines for carrying data to the hardware cells." This definition is not correct and leaves out numerous requirements. First, Ricoh's definition incorporates the same incorrect statements about the "data paths" from the definition of claim 15, which I explained in Paragraph 68. Second, Ricoh's definition also fails to limit how the "mapping" is performed. Specifically, that the mapping must be accomplished using a rule-based expert system ('432 patent's file history: April Amendment at 10 [Attachment 3]). Third, Ricoh's definition is contrary to the ordinary meaning of "expert system" and the statements in the '432 patent's file history because it fails to require that the applying of the "set of data path rules" in the knowledge base of the rule-based expert system is performed by that rule-based expert system's inference engine. Fourth, Ricoh's definition fails to require that the "rules" are stored in the knowledge base of the rule-based expert system.

V. **"generating control paths for the selected integrated circuit hardware cells"**

71. The meaning of this phrase is: "producing the necessary structural descriptions of the control paths for the selected hardware cells." This definition is consistent with the ordinary meaning and is also consistent with the '432 patent specification ('432 patent: Abstract; Figs. 1b & 13-15; 1:17-37; 2:40-42; 3:59-65; 4:39-43; 4:63-65; 5:3-12; 5:30-36; 6:18-27; 11:49-51; 13:51-14:3 [Attachment 2]).

72. Ricoh's definition: "producing signal lines for carrying control signals to the hardware cells" is not correct. First, Ricoh's definition ignores the claim language "for the selected integrated circuit hardware cells." Second, the processes of the '432 patent are for producing design data and not physical "signal lines for carrying control signals." As explained in Paragraphs 16-18, the '432 patent claims 13-17 encompass computer-aided design processes for designing an ASIC and not manufacturing them.

73. I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct.

Executed on July 13, 2004, at Summit, New Jersey.

/s/ Thaddeus J. Kowalski
Thaddeus J. Kowalski